Markt&Technik WORKSHOP AMIGA

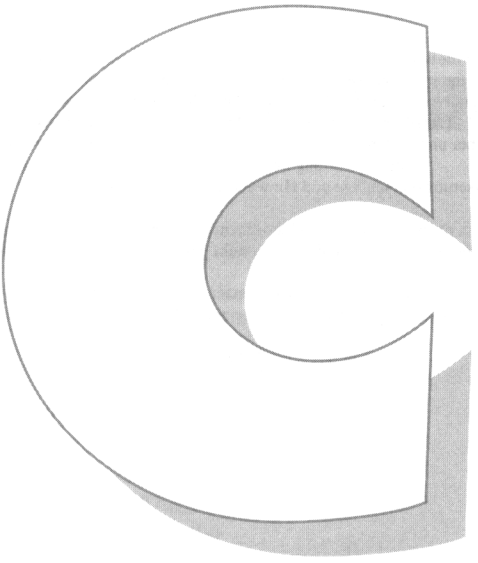
Peter Wollschlaeger

Erfolgreich starten – sicher nutzen.

WORKSHOP **AMIGA**

Herausgegeben von .TXT Redaktionsteam Baumann & Partner

Peter Wollschlaeger



Erfolgreich starten – sicher nutzen

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Wollschlaeger, Peter:
C : erfolgreich starten – sicher nutzen / Peter Wollschlaeger. –
Haar bei München : Markt-und-Technik-Verl., 1991
(Markt-&-Technik-Workshop Amiga)
ISBN 3-87791-026-2

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Amiga ist ein eingetragenes Warenzeichen der Commodore-Amiga Inc., USA
GFA-Basic ist ein eingetragenes Warenzeichen der GFA-Systemtechnik GmbH, Düsseldorf

15 14 13 12 11 10 9 8 7 6 5 4 3 2

94 93 92 91

ISBN 3-87791-026-2

© 1991 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Herstellung: Werner Leidl
Lektorat: TXT-Redaktionsteam Baumann & Partner
Dieses Produkt wurde mit Desktop-Publishing-Programmen erstellt
und auf der Linotronic 300 belichtet.
Druck: Paderborner Druck Centrum
Printed in Germany

Inhaltsverzeichnis

Vorwort 9

Kapitel 1: Einführung 11

1.1 C und der Amiga 11
1.2 Wie Sie mit diesem Buch arbeiten 13
1.3 Schreibweisen 15
1.4 Hard- und Software-Anforderungen 16
1.5 Die Installation der Compiler 18
1.5.1 Anfertigen von Sicherungs- und Arbeitskopien 19
1.5.2 Installation von Aztec C auf der Festplatte 20
1.5.3 Installation von Aztec C für zwei Diskettenlaufwerke 21
1.5.4 Installation von Aztec C für ein Diskettenlaufwerk 23
1.5.5 Installation von Lattice C auf der Festplatte 25
1.5.6 Installation von Lattice C für zwei Diskettenlaufwerke 25
1.5.7 Installation von Lattice C für ein Diskettenlaufwerk 27
1.6 Grundlagen im Umgang mit C-Compilern 29
1.6.1 Die Wahl des Editors 29
1.6.2 Kompilieren und Testen mit Aztec C 31
1.6.3 Kompilieren und Testen mit Lattice C 32
1.6.4 Wenn etwas nicht funktioniert 33

Kapitel 2: Tutorium 35

2.1	Erste Sitzung: Anatomie eines C-Programms	35
2.2	Zweite Sitzung: C-Spezialitäten für die Amiga-Programmierung	53
2.3	Dritte Sitzung: Erzeugen eines leeren Windows	70
2.4	Vierte Sitzung: Den Programmfluß kontrollieren	84
2.5	Fünfte Sitzung: Ein-/Ausgabe und das Malprogramm	98
2.6	Sechste Sitzung: Unser Window bekommt Menüs	113
2.7	Siebte Sitzung: Gadgets für die Mausklicks	128
2.8	Achte Sitzung: Requester und Alerts	145
2.9	Neunte Sitzung: Und nun wird gemalt	158
2.10	Zehnte Sitzung: Daten von und zur Disk	174

Kapitel 3: Know-how 191

3.1	Häufige Fehler und Lösungswege	191
3.2	Details zum Aztec-Compiler und Linker	196
3.3	Details zum Lattice-Compiler und Linker	198
3.4	(Überlebens-)Regeln und Tips für Programmierer	200
3.5	Kniffe, Tips und Spezialitäten	207
3.5.1	Workbench-Programme mit Icons	208
3.5.2	Chip-Memory per eigenem Mauszeiger	209
3.5.3	Daten ins Chip-Memory legen mit AllocRemember()	213
3.5.4	Zeichensätze, Größen und Stilarten	215
3.5.5	So schreibt man kompakte Programme	222
3.5.6	Compiler/Linker-Aufruf im Quelltext	224
3.5.7	Trickreiche Makros	225
3.6	Zusatz-Hardware und andere Empfehlungen	225

Kapitel 4: Referenz 229

4.1	Die wichtigsten Standard-C-Funktionen	229
4.2	Die wichtigsten Amiga-Funktionen	230
4.2.1	Die wichtigsten Funktionen von Intuition	230
4.2.2	Die wichtigsten Grafik-Funktionen	236
4.2.3	Die wichtigsten DOS-Funktionen	238

4.3	Die wichtigsten Amiga-Datenstrukturen	240
4.4	Schnellinformation: Einige Standard-Lösungen in C	247
4.4.2	Das Mehrfachproblem und Lottozahlen	253
4.4.3	File-I/O in Standard-C	254
4.4.4	Die Kommandozeile, *argc* und **argv[]*	258
4.4.5	Binär-File-I/O	259
4.4.6	Dynamische Puffer	260

Anhang

Die Include-Files zum Malprogramm	263
Glossar	280
Literaturempfehlungen	287

Stichwortverzeichnis **289**

Befehlskarte

Vorwort

Workshop – die Buchreihe für den engagierten Computerbesitzer – wurde nach völlig neuem Konzept entwickelt, das sowohl die Möglichkeiten moderner Computerprogramme als auch aktuelle didaktische Methoden berücksichtigt. Die Aufgabenstellung: Nicht einfach die Funktionsvielfalt eines Programmpakets zu demonstrieren, sondern vor allem die Fähigkeit der praktischen Umsetzung zu trainieren. Was herauskam, ist ein ausgefeiltes Praxiskonzept: mit fundiertem Startwissen, illustriertem Tutorium, zahlreichen Übungsideen, nützlichen Tips und Kniffen, Fehlerarten und Lösungsvorschlägen sowie knappem Referenzteil.

Das Herzstück eines Workshops ist das Tutorial. Es ist in zehn Sitzungen organisiert und so angelegt, daß ein nützliches Programmprodukt mit allem Hintergrund-Wissen hergestellt wird. Innerhalb kürzester Zeit sind daher praktische, transferierbare Arbeitsergebnisse sichtbar. Dann – bei steigender Erfahrung – bleiben Know-how und Referenz die nützlichen Ratgeber für die tägliche Praxis. Workshop – die Buchreihe, die mit Ihrem Wissen und Erfahrungsschatz mitwächst.

Der Autor *Peter Wollschlaeger* – bekannt durch eine Fülle von Büchern zum Amiga und zu anderen Computerthemen – ist einer der bekanntesten C-Päpste. Seine vielfältigen Erfahrungen helfen Ihnen, mit diesem Buch »C« wirklich zu begreifen – und Sie haben auch noch Spaß dabei.

Wir hoffen, daß Ihnen dieses Buch gefällt und weiterhilft. Dennoch – es gibt nichts, was man nicht noch verbessern könnte. Teilen Sie uns doch bitte Ihre Meinung zu Buch und Programm mit. Sie helfen damit anderen Lesern und uns, das Workshop-Konzept noch besser nach Ihren Bedürfnissen auszurichten. Vielen Dank.

Die Herausgeber

.TXT Redaktionsteam, Christine Baumann

Kapitel 1: Einführung

C ist eine Programmiersprache wie alle anderen auch. Es nicht schwieriger als Basic oder Pascal, viele Leute verwechseln nur etwas miteinander. Einfache Aufgaben, wie sie in Basic lösbar sind, kann man in C genauso einfach programmieren. Schwierige Jobs, wie ein ganzes Betriebssystem, lassen sich in Basic nicht schreiben, wohl aber in C. Ergo ist nicht die Sprache schwieriger, sondern es sind die Aufgaben, die damit gelöst werden können.

1.1 C und der Amiga

Daraus könnten Sie folgern, daß es sich nicht lohnt, C zu lernen, wenn man nicht vorhat, so schwierige Dinge anzugehen. Sie hätten recht, wenn es da nicht noch den Amiga gäbe. Dessen Betriebssystem ist (hauptsächlich) in C geschrieben worden und – viel wichtiger – auch sein API. API heißt »Application Program Interface«, zu deutsch, die Schnittstelle für Anwenderprogramme.

Um typische Amiga-Programme mit Windows, Pull-down-Menüs und Requestern zu schreiben oder um die Grafik-Fähigkeiten des Amiga zu nutzen, muß man über das API auf die (im ROM) eingebauten Funktionen zugreifen. Da diese Funktionen in C geschrieben wurden, kann man auch in C am besten mit ihnen umgehen, so, wie man mit einem Engländer am besten englisch spricht. C ist sozusagen die Muttersprache des Amiga.

Andere Sprachen wie Basic können das API zwar auch ansprechen, benötigen dafür aber einen Übersetzer, hier von Basic nach C. Diese

Übersetzer können aber auch nicht alles – nicht für jede API-Funktion gibt es einen Basic-Befehl – und zwingen somit den Anwender, selbst als Dolmetscher tätig zu werden. Solche Leute müssen bekanntlich zwei Sprachen beherrschen, hier also Basic und C.

C an sich

C an sich ist eine sehr einfache Sprache, sie kennt gerade ein gutes Dutzend Befehle. Dazu kommen allerdings Hunderte von Funktionen – praktisch auch Befehle – die Sie je nach Bedarf von der Diskette nachladen können. Was C so verdächtig macht, sind zwei Eigenschaften.

Zuerst sind da die Operatoren zu nennen. Den simplen Basic-Befehl *i=i+1* können Sie so auch in C schreiben. Das können Sie aber auch auf *i+=1* und noch weiter auf *i++* kürzen (wir werden das noch üben). Viele solcher Kürzel lassen sich in eine Zeile zwängen, die dann schließlich so unübersichtlich wird, daß keiner mehr durchblickt, nach drei Tagen auch nicht mehr der Erfinder selbst. Man muß nicht so programmieren, und ich werde solche Tricks im Buch vermeiden. Verraten werde ich sie dennoch.

Die zweite Eigenschaft von C ist richtig gefährlich. Man muß zwar nicht, doch man kann in C sehr maschinennah programmieren. Wenn Sie wollen, können Sie den Amiga so total umkrempeln, daß alles viel besser oder nichts mehr geht. Damit das möglich ist, fehlen in C einige Prüfungen, die andere Sprachen vornehmen. Gute Systeme geben zwar bei vielen Fehlern – längst nicht bei allen – sogenannte Warnmeldungen aus, doch auch die können Sie einfach ignorieren. Anders ausgedrückt: Sie haben in C alle Freiheiten, aber dafür tragen Sie auch selbst die volle Verantwortung. Noch ein Gleichnis: Einen Pascal-Programmierer nimmt die Mama an die Hand und paßt auf, daß ihm nichts passiert. Als C-Programmierer dürfen Sie alleine und sogar bei Rot über die Straße gehen.

Compiler, Linker und noch ein paar Begriffe

Grundsätzlich versteht ein Computer weder Basic, C noch Pascal und auch nicht Assembler. Er versteht überhaupt keine Programmiersprache, sondern nur Zahlenkolonnen. Diese Zahlen sind codierte Befehle, die sogenannte Maschinensprache. Da sich damit sehr schlecht arbeiten läßt, hat man die Hochsprachen erfunden, wozu auch C zählt. Eine Hochsprache hat wie jede andere Sprache Vokabeln und Regeln. Im Prinzip ist die Sprache C englischer Klartext. Der Unterschied ist lediglich, daß es nur wenige Vokabeln gibt und daß die Regeln absolut zu beachten sind.

Ein C-Programm wird in dieser Hochsprache formuliert, mit einer ganz normalen Textverarbeitung in den Computer eingetippt und schließlich auf der Diskette gespeichert.

Jetzt muß dieser Text in die Maschinensprache – diese Zahlenkolonnen – übersetzt werden. Diese Aufgabe übernimmt ein Programm mit dem Namen Compiler. Das Ergebnis, auch Objekt-Code genannt, wird gleichfalls auf der Disk gespeichert.

Wie schon gesagt, ist C an sich eine sehr einfache Sprache, die gerade ein gutes Dutzend Befehle kennt. Dazu kommen allerdings Hunderte von Funktionen – praktisch auch Befehle –, die zusammen mit dem Compiler geliefert werden. Einige davon – je nach Fall immer andere – braucht jedes C-Programm. Deshalb müssen diese Funktionen und der eben mit dem Compiler erzeugte Objekt-Code zusammengebunden, d.h. gemeinsam in einer Datei gespeichert werden. Genau diese Aufgabe übernimmt noch ein Programm mit dem Namen Linker. Das Ergebnis, jetzt also die dritte Datei, ist das fertige Programm.

1.2 Wie Sie mit diesem Buch arbeiten

Das Buch besteht im wesentlichen aus vier Teilen: der Einführung, die Sie gerade lesen, dem Tutorium, das einen kompletten C-Kurs beinhaltet, dem Know-how-Teil, der Ihr C-Wissen erweitert, und der Referenz, die das Wichtigste zusammenfaßt.

Einführung

Schon der Einführungsteil zeigt das grundlegende Konzept dieses Buchs. Nach einigen grundlegenden Informationen, die Sie für die weitere Arbeit brauchen, geht es sofort in die Praxis. Hier werden die Installation, der Test und die Bedienung der Software geschildert. Die Installation wird sehr ausführlich beschrieben, weil ein C-Entwicklungssystem ziemlich komplex ist. Es reicht bei weitem nicht, nur die Disketten zu kopieren.

Tutorium

Das Herzstück des Buchs bildet das Tutorium. Hier lernen Sie in zehn Sitzungen, wie man den Amiga in C programmiert. Das Prinzip lautet auch hier »nach einigen grundlegenden Informationen in die Praxis«. Daß dies nicht so trocken und langweilig wie in einem Lehrbuch erfolgt, zeigt schon die Auswahl des Beispielprogramms: Ein kleines, aber feines Malprogramm, das alle Amiga-typischen Merkmale wie Windows, Gadgets und Requester aufweist – ein Programmgerüst, das sich mit geringem Aufwand auf beliebige Aufgabenstellungen anwenden läßt. Einige Fragen zur Selbstkontrolle und ein Übungsteil mit sinnvollen Erweiterungen schließen jede Sitzung ab.

Know-how

Im Know-how-Teil werden die Feinheiten der Compiler geschildert. Oft gemachte Fehler und was man dagegen tun kann sind auch ein Thema. Die goldenen Regeln für eine erfolgreiche C-Programmierung, Kniffe, Tips und Spezialitäten schließen diesen Teil ab.

Referenz

Im Referenz-Teil finden Sie zunächst die wichtigsten C-Funktionen zu Intuition, Graphics und DOS sowie die wichtigsten Amiga-Datenstrukturen. Dann die Schnellinformation, die nach dem Motto »Wie programmiere ich ...?« einige Standard-Lösungen in C zusammenfaßt.

Anhang

Der Anhang enthält alle Include-Files des Malprogramms, das Glossar und Literaturtips.

Befehlskarte

Die handliche Referenz für Dinge, die man oft nachschlagen muß.

Eine Bitte

Natürlich finde ich es ganz toll, wenn Sie sich gleich mit Ihrem Amiga dieses Buch gekauft haben, doch bitte pausieren Sie an dieser Stelle. Bevor Sie C lernen, sollten Sie mit Ihrem Amiga vertraut sein. Sie müssen die Workbench und – besonders wichtig – das CLI bzw. die Shell kennen. Sie sollten wissen, wofür die Startup-Sequence gut ist, Directories (Inhaltsverzeichnisse) anlegen und Dateien kopieren können. Alle diese Dinge sind in den zum Amiga gehörenden Handbüchern gut beschrieben.

1.3 Schreibweisen

Es folgen jetzt leider einige Begriffe, die erst später erklärt werden, doch das Thema »Schreibweisen« muß hier schon angesprochen werden, damit Sie beim Weiterlesen wissen, was das soll. Im laufenden Text – nicht in den Listings – werden Bezeichner der Sprache C fast immer kursiv geschrieben. Das gilt für

- Variablennamen, z.B. *zaehler, wert1*
- Funktionsnamen, z.B. *printf(), ClearMenuStrip()*
- Direktiven, z.B. *#define, #include*
- Groß (versal) geschrieben werden Flags und nicht zum Standard-C gehörende Makros, z.B. DELTAMOVE, BYTE.

Absätze können mit einem Hinweis- oder mit einem Gefahrensymbol gekennzeichnet sein.

☞ Ein Hinweis ist eine Empfehlung oder ein Rat. Wenn Sie ihn ignorieren oder vergessen, passiert nicht allzuviel. Im schlimmsten Fall meldet der Compiler einen Fehler, den Sie korrigieren müssen.

⚠ **Wenn dieses Symbol auftaucht, ist Gefahr im Verzug. Es gibt zum Beispiel Tippfehler, die der Compiler nicht erkennt. Bei solchen Fehlern läuft das Programm falsch oder stürzt ab.**

1.4 Hard- und Software-Anforderungen

Programmentwickler unterscheiden immer zwei Computer-Systeme, nämlich das Entwicklungssystem – der Rechner, auf dem die Programme entstehen – und die Zielmaschine. Letztere ist der Computer, auf dem die fertigen Programme laufen sollen. Grundsätzlich gilt, daß der Entwicklungsrechner mindestens alle Merkmale der Zielmaschine haben sollte, doch das ist nur die halbe Wahrheit. Es nützt nichts, eine kleine Zielmaschine zu definieren, nach dem Motto »weil ich nur einen kleinen Amiga habe, schreibe ich nur kleine C-Programme«. Die Compiler selbst brauchen eine gewisse Rechneraustattung, wenn sie vernünftig arbeiten sollen. Fehlen diese Voraussetzungen, wird so ein System umständlich bedienbar und langsam. Ein Profi kann sich da notfalls noch besser helfen als ein Einsteiger – ich habe auch schon mal ein paar Tage mit einem Minimum-Amiga gearbeitet, weil mein 2000er zur Reparatur war –, doch praktisch ergibt sich daraus eine seltsame Konsequenz:

Gerade als Einsteiger sollten Sie nicht mit einer zu schmalen Hardware-Basis starten. Sie haben genug damit zu tun, C zu lernen. Wenn Sie sich dann noch mit den Unzulänglichkeiten der Hardware herumärgern, ständig Diskjockey spielen und nach jedem kleinen (oder großen) Fehler ewig auf den Compiler warten müssen, dann ärgern Sie sich mehr, als daß Sie C lernen.

Hardware-Voraussetzungen

Generell arbeiten die in diesem Buch beschriebenen C-Compiler von Aztec und Lattice auf jedem Amiga 500, 1000, 2000 und 2500. Auf einem 3000er habe ich sie nicht getestet.

Die ideale Hardware-Basis für diese C-Systeme ist eine Festplatte und wenigstens 1 Mbyte RAM. Damit kann man sehr schön arbeiten.

Wenn Sie es ganz eilig haben und sehr große Programme entwickeln wollen, müssen Sie den Compiler nebst »Zubehör« im RAM halten. Dafür braucht man bei Lattice C in der höchsten Ausbaustufe 2,5 Mbyte. Es wird oft empfohlen –, auch von Lattice – bei Geldmangel eher eine RAM-Aufrüstung als eine Festplatte zu kaufen. Dem kann ich nicht ganz zustimmen, denn es dauert »ewig«, bis das ganze System von den Disketten in den RAM-Speicher geladen wird, und nach so manchem Absturz wird diese Übung immer wieder fällig. Bei den kleineren und mittleren Programmen dieses Buchs ist die Festplatten-Lösung im Endergebnis schneller.

Mit zwei Diskettenlaufwerken kann man durchaus arbeiten, dann reichen sogar 512 Kbyte RAM. Genau das ist die von Aztec und Lattice genannte Minimal-Konfiguration. Sie sollten jedoch bedenken, daß für jeden Durchlauf diverse Programme und Daten (Editor, Compiler, Linker, Include-Files und Libraries) immer wieder neu von den Disketten geladen werden müssen. Da ist es dann sehr praktisch, wenn man wenigstens 1 Mbyte RAM hat, so daß einige dieser Dateien im Speicher gehalten werden können.

Rechner mit nur einem Diskettenlaufwerk werden von Aztec und Lattice offiziell nicht unterstützt. Die mitgelieferten Installationsprogramme kennen diesen Fall nicht. Ich zeige dennoch, wie Sie die Software für diesen Fall installieren können. Diese Lösungen sind aber nur für die paar Tage gedacht, die Sie brauchen, um Ihren Amiga aufzurüsten. Wenn Sie nicht viel mehr als 250,-- DM ausgeben wollen, empfehle ich ein zweites Diskettenlaufwerk.

Software-Voraussetzungen

Amiga-seitig benötigen Sie:

- Kickstart 1.3
- Workbench 1.3

Als Compiler empfehlen wir:

- Aztec C der Version 5.0.x

oder

- Lattice C der Version 5.0.x

In beiden Fällen reicht auch die sogenannte kleine Version. Die großen Versionen sollten Sie nicht ohne eine Festplatte einsetzen. Beachten Sie, daß bei Aztec C die kleine Version »Professional-System« und die große »Developer-System« heißt.

☞ Wenn Sie noch keinen dieser beiden Compiler besitzen, lesen Sie dieses ganze Kapitel durch. Es wird Ihrer Entscheidungsfindung nützen.

Das Buch setzt einen dieser beiden Compiler voraus. Beide haben ihre kleinen Vor- und Nachteile, ich kann jedoch beim besten Willen nicht sagen, daß der eine besser ist als der andere. Eines bleibt jedoch festzustellen: Wenn der Rechner nur Diskettenlaufwerke – vielleicht sogar nur

eines – und wenig RAM-Speicher hat, kommt man mit Aztec C besser zurecht. Lattice hat den besseren Editor und auch die besseren Handbücher.

Grundsätzlich sind auch andere Compiler einsetzbar, sofern diese den ANSI-Standard erfüllen. Das Buch erläutert jedoch nur die Bedienung von Aztec C und Lattice C der Version 5.0.x. Auf Unterschiede zu den Vorgänger-Versionen wird stellenweise hingewiesen.

Ich empfehle jedoch dringend, eine 5er-Version einzusetzen. Nur die Versionen 5.0.x und höher erfüllen voll den sogenannten ANSI-Standard (eine anerkannte amerikanische Norm) und stellen damit ein sehr modernes C dar. Natürlich sollten Sie, wenn schon, dann auch das neuste C lernen, aber »ANSI« hat noch zwei Vorteile.

1. Programme für den alten K&R-Standard laufen auch unter ANSI C, umgekehrt gilt das nicht (auf die Unterschiede wird in der ersten Sitzung noch eingegangen).
2. Auch die guten C-Compiler anderer Rechner halten sich an den ANSI-Standard, Sie können also leicht C-Programme oder Listings auf den Amiga portieren.

1.5 Die Installation der Compiler

Die ersten Schritte sind für beide Compiler die gleichen, nämlich das Anlegen von Sicherungs- und Arbeitskopien. Das sind zwei verschiedene Dinge.

Zuerst kopieren Sie Ihre kostbaren Originale 1:1, das ergibt die Sicherungskopien. Danach legen Sie die Originale gut weg und arbeiten nur noch mit den Sicherungskopien weiter.

Als Festplattenbesitzer installieren Sie dann von den Sicherungskopien aus das System auf der Festplatte. Die Disketten brauchen Sie erst wieder, wenn ein Fehler auf der Festplatte zu beheben ist oder Sie Änderungen rückgängig machen wollen.

Wenn Sie nur Diskettenlaufwerke haben, müssen Sie die Sicherungskopien nochmals duplizieren. Das ergibt dann die Arbeitskopien. Dieser Umstand ist erforderlich, weil die Arbeitskopien geändert werden und somit mit den Sicherungskopien nicht mehr übereinstimmen.

☞ Wenn Sie Aztec C auf einem Amiga mit zwei Diskettenlaufwerken einsetzen wollen, brauchen Sie von den Disketten »Aztec1« und »Aztec2« keine Arbeitskopien zu erstellen, das erledigt das Installationsprogramm.

1.5.1 Anfertigen von Sicherungs- und Arbeitskopien

Wenn Sie eine Lattice-Diskette einlegen, wird sie sich mit einem riesigen Disk-Icon auf der Workbench breitmachen. Leider hat das auch zur Konsequenz, daß Sie die Disketten nicht auf der Workbench duplizieren können. Aztec hat das Problem nicht. Dennoch wechseln Sie in beiden Fällen in die Shell oder in das CLI, indem Sie das Shell- oder das CLI-Icon doppelt anklicken. Alle weiteren Operationen laufen sowieso unter dieser Umgebung. Für jede Diskette wiederholen Sie die folgenden Schritte:

Stellen Sie sicher, daß auf den Originalen der Schreibschutz eingeschaltet ist. Geben Sie ein:

Mit zwei Laufwerken:

```
diskcopy df0: df1: Return
```

Mit einem Laufwerk:

```
diskcopy df0: df0: Return
```

Folgen Sie den Anweisungen auf dem Schirm. In diesem Dialog bedeuten:

- Source (From Disk): die Diskette, die kopiert werden soll.
- Destination (To Disk): die Diskette, auf die kopiert werden soll.

☞ Der Diskcopy-Befehl kopiert auch die Namen der Disketten 1:1. Bitte ändern Sie diese Namen nicht. Sowohl während der Installation als auch im Betrieb wird auf diese Namen Bezug genommen.

☞ Die folgenden sechs Installations-Prozeduren stellen einige, aber nicht alle denkbaren Möglichkeiten dar. Sie sollten deshalb, nachdem Sie Ihren Compiler für Ihre Hardware installiert haben, sich einmal die anderen Konfigurationen anschauen. Vielleicht können Sie davon auch noch etwas gebrauchen.

1.5.2 Installation von Aztec C auf der Festplatte

Die Prozedur ist sehr einfach, hier das Kochrezept:

1. Starten Sie Ihren Amiga und wechseln Sie (wenn nicht schon geschehen) auf die Festplatte.
2. Legen Sie die Kopie von »Aztec1« in das Laufwerk DF0: ein.
3. Gehen Sie in das CLI und tippen Sie ein:

   ```
   df0:hdinstall Return
   ```

4. Im Window SELECT LIBRARIES klicken Sie mit der Maus an:

   ```
   X  Small Code/Small Data
   X  32-bit Integers
   X  MANX IEEE
   ```

5. Folgen Sie dem Dialog, der Sie im wesentlichen auffordert, nacheinander die anderen Disketten einzulegen.

Jetzt gibt es zwei Möglichkeiten:

1. Sie haben noch andere Compiler auf der Festplatte, die auch Umgebungsvariablen wie LIB und INCLUDE brauchen. In diesem Fall legen Sie im s-Directory Ihrer Startdiskette bzw. bei einer Autoboot-Festplatte dort eine Textdatei dieses Inhalts an:

   ```
   path "dh0:Aztec/Bin"
   mset "CCTEMP=ram:"
   mset "CLIB=dh0:Aztec/Lib"
   mset "INCLUDE=dh0:Aztec/Include"
   ```

☞ Den im Handbuch häufig erwähnten set-Befehl gibt es nicht. Statt dessen müssen Sie mset (MANX's set) so einsetzen, wie es die obigen Beispiele zeigen.

Wenn Sie den Z-Editor zusammen mit der Quickfix-Compiler-Option nutzen wollen (siehe 1.6.1, Z-Editor), fügen Sie noch diese Zeile ein:

```
mset "CCEDIT=dh0:Aztec/bin/z"
```

Speichern Sie diese Datei unter dem Namen *az*. Später werden Sie in das Aztec-Directory wechseln und dann *execute az* tippen, womit Sie die Umgebungsvariablen von Aztec C definieren. Diese braucht das System, um die einzelnen Dateien zu finden.

2. Aztec C ist Ihr einziger Compiler auf der Festplatte. In diesem Fall reicht es, die obigen vier Zeilen in Ihre Startup-Sequence einzutragen.

1.5.3 Installation von Aztec C für zwei Diskettenlaufwerke

Vorab: Zumindest bei meiner im September 1990 erworbenen Version 5.0b von Aztec C stimmt die tatsächliche Installations-Prozedur nicht mit der im Handbuch überein. Die beiden ersten Disketten (Aztec1 und Aztec2) sind zwar das lauffähige Entwicklungssystem, nur die Startup-Sequence paßt dafür überhaupt nicht, und die Disketten sind voller als für den echten Betrieb nötig. Deshalb sollten Sie die mitgelieferte Installations-Prozedur anwenden und erst danach (oder später) mit der Maßschneiderei beginnen, wie sie im Handbuch erwähnt wird.

Hier das Kochrezept für die richtige Installation:

1. Halten Sie zwei leere Disketten bereit, diese müssen nicht formatiert sein.

2. Legen Sie die Kopie von »Aztec1« in Ihr Startlaufwerk (DF0:) ein. Wenn Ihr Amiga schon läuft, gehen Sie in das CLI und tippen ein:

   ```
   execute df0:install
   ```
   [Return]

   Ist Ihr Amiga noch aus, starten Sie ihn einfach mit der Diskette »Aztec1« im Startlaufwerk (DF0:).

3. Auf die Frage »install to hard disk?« antworten Sie mit »n [Return]«.

4. Auf die Frage »do you wish to read the 'read.me file'?« können Sie mit »n [Return]« antworten.

5. Auf die Frage »is your second floppy drive called df1:?« antworten Sie mit »y [Return]«, wenn Ihr zweites Laufwerk eingebaut ist. Heißt Ihr zweites Laufwerk »DF2:« (typisch für ein externes Laufwerk), antworten Sie »n [Return]«.

6. Eine Aufforderung wie

   ```
   Insert a blank disk in drive dfx:
   then press <enter> and to continue
   ```

   heißt, daß Sie eine leere Diskette in das Laufwerk dfx: (DF1: oder DF2:) einlegen und dann die [Return]-Taste drücken sollen.

7. In dem dann erscheinenden Window SELECT LIBRARIES klicken Sie mit der Maus an:

```
X  Small Code/Small Data
X  32-bit Integers
X  MANX IEEE
```

Nun folgen Sie dem weiteren Dialog. Eine Aufforderung wie

```
Insert Aztec Disk #2 in df0:
Click 'OK' to continue
```

heißt, daß Sie die zweite Aztec-Diskette (Ihre Kopie) in das Laufwerk DF0: einlegen und dann mit der Maus auf das OK-Feld klicken sollen.

Umstellung auf die deutsche Tastatur

Wenn Sie mit einer Kopie der ersten Diskette (Aztec1) booten, wird die amerikanische Tastaturbelegung eingestellt. Um das zu ändern, sind zwei Maßnahmen erforderlich.

Im Directory *:devs/keymaps* löschen Sie die Datei *usa1* und kopieren dafür die Datei *d* von Ihrer Workbench-Diskette dorthin.

Mit einem Editor (zum Beispiel ED) ändern Sie die Datei *startup-sequence* im s-Directory. Fügen Sie diese Zeile ein:

```
:System/setmap d
```

Wenn Sie mehr als 512 Kbyte RAM haben, sollten Sie diesen nutzen. Um beispielsweise die Libraries und alle Programme im C-Directory (Compiler und Linker inklusive) in den RAM-Speicher zu bringen, muß der entsprechende Teil der Startup-Sequence so aussehen:

```
copy df1:libs ram:
mset "CLIB=ram:"
copy df0:c ram:
path ram: add
```

Prüfen Sie bitte, auf welchen Disketten sich tatsächlich die Directories *libs* und *c* befinden, und setzen Sie die entsprechenden Namen oder Laufwerke ein.

☞ Eine alternative Lösung ist mittels »resident« ab der Workbench 1.3 möglich. Die Startup-Sequence im Abschnitt 1.5.7 zeigt die Anwendung.

1.5.4 Installation von Aztec C für ein Diskettenlaufwerk

Die Arbeit ist nicht schwierig, aber langwierig. Beginnen Sie mit der Arbeitskopie von »Aztec1« und löschen Sie alles, was nicht im folgenden Directory-Listing steht. Dann kopieren Sie alles, was noch fehlt, von »Aztec2« auf »Aztec1«.

```
c (dir)
       AddBuffers          as
       Assign              cc
       CD                  Copy
       Delete              Dir
       Echo                ed
       endcli              Execute
       ln                  loadwb
       Makedir             mset
       Path                Run
       Stack
     System (dir)
       CLI                 SettMap
     l (dir)
       Disk-Validator      Newcon-Handler
       Port-Handler        Ram-Handler
     devs (dir)
          keymaps (dir)
            d
          clipboards (dir)
       system-configuration
     s (dir)
       startup-sequence
     t (dir)
     fonts (dir)
     /* unverändert übernommen */
     libs (dir)
       c.lib               diskfont.library
       icon.library        info.library
       version.library
     include (dir)
     /* unverändert übernommen */
     Empty (dir)
```

Listing 1.5.1: Das Inhaltsverzeichnis von Aztec C

Nun müssen Sie noch die Startup-Sequence im s-Directory mit einem Editor (z.B. ED) so ändern, daß sie mindestens wie folgt aussieht:

```
addbuffers df0: 10
echo "Aztec wird vorbereitet, bitte warten..."
Stack 8000
path ram:
assign ENV: ram:
path Aztec1:system add
mset "CLIB=Aztec1:libs"
mset "INCLUDE=Aztec1:include"
mset "CCTEMP=ram:"
setmap d
echo "Fertig"
```

Wenn Sie den Z-Editor zusammen mit der Quickfix-Compiler-Option nutzen wollen (siehe 1.6.1, Z-Editor), fügen Sie noch diese Zeile ein:

```
mset "CCEDIT=Aztec1:c/z"
```

und kopieren auch noch das Z-Programm in das c-Directory.

Das Beispiel unterstellt, daß die Diskette »Aztec1« heißt. Sollten Sie den Namen geändert haben, müssen Sie die Startup-Sequence anpassen. Das Ergebnis ist eine bootfähige Diskette, auf der sich alles befindet, was Sie für die C-Entwicklung vorerst benötigen. Wie schon gesagt: Diese Lösung ist nur für die paar Tage gedacht, die Sie brauchen, um Ihren Amiga aufzurüsten.

Wenn Sie 1 Mbyte RAM und mehr haben

Wenn Sie 1 Mbyte RAM (oder mehr) haben, können Sie Teile des Systems in den RAM-Speicher verlagern. Das folgende Beispiel ersetzt die Zeile *mset "CLIB=Aztec1:libs"*. Entsprechend können Sie – wenn der RAM-Speicher noch reicht – mit dem INCLUDE-Directory verfahren.

```
copy df0:libs ram:
mset "CLIB=ram:"
copy df0:c ram:
path ram: add
```

1.5.5 Installation von Lattice C auf der Festplatte

Die Prozedur ist sehr einfach, hier das Kochrezept:

1. Starten Sie Ihren Amiga und wechseln Sie (wenn nicht eh schon da) auf die Festplatte.
2. Legen Sie die Kopie von »Lattice_C_5.0.1« in das Laufwerk DF0: ein.
3. Gehen Sie in das CLI und tippen Sie ein:

   ```
   execute df0:s/install_hd
   ```
   Return

4. Folgen Sie dem Dialog, der Sie auffordert, nacheinander die anderen Disketten einzulegen.

Jetzt gibt es zwei Möglichkeiten:

1. Sie haben noch andere Compiler auf der Festplatte, die auch Umgebungsvariablen wie LIB und INCLUDE brauchen. In diesem Fall legen Sie im s-Directory Ihrer Startdiskette bzw. bei einer Autoboot-Festplatte dort eine Textdatei dieses Inhalts an:

   ```
   assign LC: sys:lc/c
   assign INCLUDE: sys:lc/include
   assign LIB: sys/lc/lib
   assign QUAD: ram:
   ```

   Speichern Sie diese Datei unter dem Namen *lat*. Später werden Sie in das LC-Directory wechseln und dann *execute lat* tippen, womit Sie die Umgebungsvariablen von Lattice C definieren. Diese braucht das System, um die einzelnen Dateien zu finden.
2. Lattice C ist Ihr einziger Compiler auf der Festplatte. In diesem Fall reicht es, die obigen vier Zeilen in Ihre Startup-Sequence einzutragen.

1.5.6 Installation von Lattice C für zwei Diskettenlaufwerke

Die beiden ersten Disketten (Lattice_C_5.0.1 und Lattice_C_5.0.2) sind schon das lauffähige Entwicklungssystem. Lattice_C_5.0.1 gehört in das Startlaufwerk, Lattice_C_5.0.2 in das zweite. Wenn Sie damit Ihren Amiga starten, werden automatisch die Umgebungsvariablen korrekt gesetzt.

Damit Sie wenigstens noch vorerst Ihre Programme auf den zu 99 Prozent vollen Disketten unterbringen können, müssen Sie einige Dateien löschen (bitte nur auf den Arbeitskopien). Auf der Disk 1 (Lattice_C_5.0.1) bieten sich dafür an:

```
sys:     #?info, "on Lattice C", read.me
c:       Avail, List, Queue, RemRad, Rename;
         weitere CLI-Befehle, die Sie
         selten brauchen.
System:   format, Diskcopy, #?info.
devs:    narrator.device, serial.device
l:       FastFileSystem
libs:    translator-library
```

Auf der Disk 2 (Lattice_C_5.0.2) können Sie im Lib-Directory diese Mathematik-Libraries löschen:

```
lcmieeelib
lcmffp.lib
lcm881.lib
```

Reicht das noch nicht, löschen Sie auch die beiden Debug-Libraries und dann weiter nach Gutdünken. Fehlt Ihnen plötzlich eine Library, wird das der Linker schon melden. Dann müssen Sie diese eben wieder auf die Disk kopieren und dafür eine andere löschen. Ansonsten empfiehlt sich, eine dritte formatierte Diskette bereitzuhalten, auf der Sie Ihre Quelltexte und Programme speichern.

Umstellung auf die deutsche Tastatur

Wenn Sie mit einer Kopie der ersten Diskette (Lattice_C_5.0.1) booten, wird die amerikanische Tastaturbelegung eingestellt. Um das zu ändern, sind zwei Maßnahmen erforderlich.

Im Directory *:devs/keymaps* löschen Sie die Datei *usa1* und kopieren dafür die Datei *d* von Ihrer Workbench-Diskette dorthin.

Mit einem Editor (zum Beispiel ED) ändern Sie die Datei *startup-sequence* im s-Directory. Fügen Sie als zweite Zeile ein:

```
:System/setmap d
```

Im Editor LSE bleiben dennoch Ctrl + y und Ctrl + z vertauscht. Sie müssen daher eine Zeile mit Ctrl + z löschen.

Wenn Sie 1 Mbyte RAM und mehr haben

Wenn Sie einen Amiga mit 1 Mbyte oder mehr RAM haben, fügen Sie in der Datei *install_floppy* im s-Directory diese Zeilen hinzu:

```
resident lc:lc1
resident lc:lc2
resident lc:blink
```

Die Texte stehen bereits in Echo-Anweisungen und müssen nur noch geändert werden. Sie erreichen damit, daß der Compiler und der Linker im RAM gehalten und nicht jedesmal neu von der Diskette geladen werden müssen.

Wenn Sie einen Amiga mit 2 Mbyte oder mehr RAM-Speicher haben, ersetzen Sie in der Datei *install_floppy* im s-Directory die Zeile

```
assign LIB:      Lattice_C_5.0.2:lib
```

durch diese beiden Zeilen:

```
copy Lattice_C_5.0.2:lib ram:
assign LIB:  ram:
```

Damit werden alle Libraries in die RAM-Disk kopiert, womit das System drastisch schneller wird.

1.5.7 Installation von Lattice C für ein Diskettenlaufwerk

Die beiden ersten Disketten (Lattice_C_5.0.1 und Lattice_C_5.0.2) sind schon das lauffähige Entwicklungssystem. Lattice_C_5.0.1 gehört in das Startlaufwerk, Lattice_C_5.0.2 normalerweise in das zweite Laufwerk. Wenn letzteres fehlt, können Sie das System nur so konfigurieren, daß die Programme immer abwechselnd die eine oder die andere Diskette anfordern. Dafür müssen Sie die Startup-Sequence im s-Directory mit einem Editor (z.B. ED) so ändern, daß sie in etwa wie folgt aussieht:

Auf einem Amiga mit nur 512 Kbyte RAM lassen Sie alle Resident-Befehle weg.

```
echo "Lattice C wird vorbereitet, bitte warten..."
c:resident sys:c/resident
resident sys:c/assign
resident sys:c/echo
resident sys:c/path
resident sys:c/dir
resident sys:c/MakeDir
resident sys:c/cd

c:SetPatch >NIL:
c:AddBuffers df0: 30
c:FF >NIL: -0
c:cd ram:
c:makedir ram:env
c:makedir ram:clipboards

assign CLIPS: ram:clipboards
assign ENV: ram:env
assign QUAD:    RAM:
assign LC:      Lattice_C_5.0.1:c
assign INCLUDE: Lattice_C_5.0.2:CompactH
assign LIB:     Lattice_C_5.0.2:lib

path lc: add
resident lc:lc1
resident lc:lc2
resident lc:blink

cd df0:
:system/setmap d
echo "Fertig"
```

Listing 1.5.2: Startup-Sequence

Damit Sie wenigstens noch vorerst Ihre Programme auf den zu 99 Prozent vollen Disketten unterbringen können, müssen Sie einige Dateien löschen (bitte nur auf den Arbeitskopien). Dazu verfahren Sie wie bei der Lösung für zwei Laufwerke (Abschnitt 1.5.6).

Wenn Sie einen Amiga mit 2 Mbyte oder mehr RAM haben, ersetzen Sie die Zeile

```
assign LIB:     Lattice_C_5.0.2:lib
```

durch diese beiden:

```
copy Lattice_C_5.0.2:lib ram:
assign LIB:  ram:
```

Damit werden alle Libraries in die RAM-Disk kopiert, womit das System drastisch schneller wird und das häufige Diskettenwechseln entfällt.

1.6 Grundlagen im Umgang mit C-Compilern

Wie Sie schon gemerkt haben, war die Installation des Compilers nebst seinem Zubehör recht aufwendig, doch die Mühe hat sich gelohnt. Sie können das System bedienen, ohne sich darum kümmern zu müssen, auf welchen Disketten und in welchen Directories sich die einzelnen Programme und Daten befinden. Dennoch bleibt noch etwas zu tun.

1.6.1 Die Wahl des Editors

Wie schon gesagt, ist ein C-Programm zuerst einmal Text, den Sie mit jedem Textprogramm oder Editor (ein anderer Name) eingeben können. Dennoch werden Sie hauptsächlich im Editor arbeiten, hier gestalten Sie Ihr Programm. Daher sollten Sie schon auf ein Werkzeug Wert legen, mit dem Sie gerne arbeiten.

Die erste Regel lautet: Wenn Sie einen bestimmten Editor gewohnt sind, bleiben Sie dabei. Wenn Sie mit Ihrem Lieblings-Textverarbeitungsprogramm arbeiten wollen, ist das auch OK. Sie müssen hierbei nur darauf achten, daß der Text unformatiert (als ASCII) gespeichert wird. Im folgenden werden vier Editoren kurz vorgestellt, Sie haben die Qual der Wahl.

Der Amiga-Editor ED

Der Editor ED wird seit »Urzeiten« mit dem Amiga geliefert, ist in seinem Handbuch und in zahllosen anderen Werken beschrieben und allgemein sehr beliebt. Wenn Sie Ihre C-Programme mit Hilfe von ED erstellen wollen, ist das OK.

MEMACS

MEMACS ist ein Geschenk von Commodore (auf der Extras-Diskette), das unverdientermaßen wenig bekannt ist. MEMACS ist ein Editor für Programmierer, da er – unter vielen anderen Programmierer-Features – erlaubt, mehrere Textdateien gleichzeitig zu bearbeiten. Gerade beim Amiga ist das von besonderer Bedeutung, weil sich viele Dinge, wie der Aufbau von Windows oder Menüs, in allen Programmen wiederholen. Da ist es denn sehr praktisch, wenn man die jeweils am besten passende Routine aus verschiedenen anderen Programmen leicht ausschneiden und einsetzen kann. Die Bedienung von MEMACS ist dank der Pull-down-Menüs recht einfach. Sie wird im Amiga-DOS-1.3-Benutzerhandbuch mehr als ausführlich auf 38 Seiten beschrieben. Eines kommt jedoch nicht so klar heraus, und das ist der für Sie besonders wichtige C-Modus. Um diese Betriebsart einzuschalten, gehen Sie so vor:

- Tippen Sie [Esc] + [s] oder wählen Sie *Set* aus dem Extras-Menü.
- Es erscheint die Frage *Set what:*. Geben Sie *mode* [Return] ein.
- Es erscheint die Frage *Mode:*. Geben Sie *c* [Return] ein.

Jetzt können Sie aus dem Search-Menü *Fence-match* wählen oder [Esc]+ [Ctrl] + [f] tippen. Generell sucht *Fence-match* das nächste Auftreten des Zeichens, auf dem der Cursor gerade steht. Besonders praktisch ist das, wenn Sie von der offenen »{«-Klammer ausgehend die geschlossene Klammer »}« suchen. In C werden die geschweiften Klammern sehr häufig benötigt. Sie können damit leicht kontrollieren, ob die Klammern paarig sind – eine beliebte Fehlerquelle –, aber auch schnell vom Beginn einer Funktion an ihr Ende gelangen.

Der Z-Editor von Aztec

Der Editor mit dem kurzen Namen »Z« wird mit Aztec C geliefert. Sein Vorteil ist die QuickFix-Option. Ist diese eingeschaltet, passiert folgendes: Wenn Sie ein Programm kompilieren und dabei ein Fehler auftritt, wird automatisch »Z« gestartet, der Cursor steht in der (wahrscheinlich) fehlerhaften Zeile, und die Fehlermeldung wird angezeigt. Sie können den Fehler korrigieren, wieder den Compiler aufrufen, und das Spiel kann von vorne beginnen.

»Z« ist eine ziemlich exakte Kopie des Unix-Editors »Vi«, woraus auch sein Nachteil resultiert, nämlich die furchtbar komplizierte Bedienung, die

man auf 50 englischen Handbuchseiten regelrecht pauken muß. Wenn Sie das möchten... Doch ich kann Ihnen wirklich nicht raten, mit »Z« zu beginnen.

Der Lattice-Screen-Editor LSE

Der Lattice-Editor LSE ist im Vergleich zu »Z« eine richtige Erholung, da kommt Freude auf. Es handelt sich um einen Multi-Window-Editor (wie MEMACS) mit Pull-down-Menüs, über die alle wichtigen Funktionen erreicht werden können. Der Rest – und alles alternativ – läuft über Tastenkürzel, doch die müssen Sie auch nicht lernen. Ein Druck auf die [F1]-Taste bringt Hilfe, und mit [Esc] sind Sie wieder im Text. Mit der [F4]-Taste starten Sie den Compiler. Stellt der einen Fehler fest, wird er angezeigt, und der Cursor steht in der fehlerhaften Zeile. Um das Programm zu linken und zu starten, müssen Sie aber doch LSE verlassen.

Sie können LSE, ohne auch nur einen Blick in das Handbuch geworfen zu haben, sofort bedienen. Dennoch würde ich Ihnen raten, die rund 80 Seiten einmal durchzublättern. LSE bietet unheimlich viele Extras.

Einen Nachteil hat auch LSE. Für die Control-Codes gilt die amerikanische Tastaturbelegung. Dazu müssen Sie aber nur wissen, daß Y und Z vertauscht sind und Sie für die Blockmarkierung [Ctrl] + [ü] für den Beginn sowie [Ctrl] + [*] für das Ende tippen müssen.

1.6.2 Kompilieren und Testen mit Aztec C

Wenn Sie ein Diskettensystem gemäß Abschnitt 1.5.3 oder 1.5.4 einsetzen, reicht es, Ihren Amiga mit diesen Disketten zu starten. Die erste Diskette (Aztec1) muß im Laufwerk DF0: eingelegt sein.

Arbeiten Sie mit einer Festplatte, gehen Sie in das CLI oder die Shell und wechseln in das Aztec-Directory. Wenn Sie die path- und mset-Befehle gemäß Abschnitt 1.5.2 nicht in Ihre Startup-Sequence eingetragen hatten, geben Sie jetzt ein:

```
execute az [Return]
```

Geben Sie mit einem Editor Ihrer Wahl (siehe 1.6.1) folgenden Text ein, und speichern Sie ihn unter dem Namen *hallo.c*.

```
main()
{
  printf("Hallo Welt \n");
}
```

Wieder im CLI, kompilieren Sie diesen Quelltext mit dem Befehl

```
cc hallo Return
```

Jetzt müssen Sie noch linken, und zwar mit

```
ln hallo -lc Return
```

Damit ist das Programm erstellt. Wenn Sie jetzt *hallo* Return eingeben, sollten Sie den Text »Hallo Welt« sehen.

1.6.3 Kompilieren und Testen mit Lattice C

Wenn Sie ein Diskettensystem gemäß Abschnitt 1.5.3 oder 1.5.4 einsetzen, reicht es, Ihren Amiga mit diesen Disketten zu starten. Die erste Diskette (Aztec1) muß im Laufwerk DF0: eingelegt sein.

Arbeiten Sie mit einer Festplatte, gehen Sie in das CLI oder die Shell, und wechseln Sie in das LC-Directory. Wenn Sie die assign-Befehle gemäß Abschnitt 1.5.5 nicht in Ihre Startup-Sequence eingetragen hatten, geben Sie jetzt dieses Kommando ein:

```
execute lat Return
```

Geben Sie mit einem Editor Ihrer Wahl (siehe 1.6.1) folgenden Text ein, und speichern Sie ihn unter dem Namen *hallo.c*.

```
void main()
{
  printf("Hallo Welt \n");
}
```

Wieder im CLI, kompilieren und linken Sie diesen Quelltext mit dem Befehl

```
lc -L hallo Return
```

☞ Das L bei –L muß groß geschrieben werden.

Damit ist das Programm erstellt. Wenn Sie jetzt *hallo* Return eingeben, sollten Sie den Text »Hallo Welt« sehen.

1.6.4 Wenn etwas nicht funktioniert

Zum Schluß noch einige Tips, für den Fall, daß etwas nicht funktionieren sollte.

Stellen Sie sicher, daß die Datei *hallo.c* existiert. Testen Sie das durch die Eingabe von

```
type hallo.c [Return]
```

Wenn der Compiler einen Fehler (Error) meldet, muß ein Tippfehler vorliegen. Prüfen Sie anhand des Listings speziell:

- Folgen nach *main* die runden Klammern »()«?
- Sind die Klammern vor und nach der printf()-Zeile geschweifte Klammern (erst auf, dann zu)?
- Steht der Text innerhalb der runden Klammern von *printf()* in Anführungszeichen? Endet diese Zeile mit einem Semikolon?

Wenn Programme oder Dateien nicht gefunden werden, kann das folgende Gründe haben:

- Die Umgebungsvariablen sind nicht oder falsch eingestellt. Kontrollieren Sie die assign-, path- und mset-Befehle gemäß Abschnitt 1.5 auf Vollständigkeit und richtige Schreibweise. Sind diese Befehle nicht Teil der Startup-Sequence, müssen Sie sie selbst aktivieren (*execute az* bzw. *execute lat*).
- Programme oder Daten sind nicht oder in das falsche Verzeichnis kopiert worden. Das betrifft speziell die Ein-Disketten-Lösung von Aztec C. Kontrollieren Sie das gemäß Abschnitt 1.5.4

☞ Aus drucktechnischen Gründen wurden öfter sogenannte Stringliterale in einer Zeile begonnen und in der nächsten fortgesetzt. Das melden die Compiler als Fehler. Beachten Sie deshalb unbedingt die folgende ☞ Erläuterun g »Zeilenumbruch«

Zeilenumbruch von Stringliteralen

Ein Stringliteral ist ein Text, der in Anführungszeichen steht. Dieses Literal muß in einer Zeile stehen. Zum Beispiel ist korrekt:

```
printf("Das ist ein String-Literal");
```

Hingegen ist nicht korrekt

```
printf("Das ist ein
        String-Literal");
```

Die zweite Form haben wir aus drucktechnischen Gründen manchmal wählen müssen. Wir meinen, das ist günstiger als eine zu kleine Schrift. Der Lattice-Compiler meldet in diesem Fall

```
String too long or not terminated
```

Aztec hingegen meldet dann gleich vier Fehler, nämlich

```
unterminated string
need right parenthesis or comma in arg list
missing semikolon
unterminated string
```

Schreiben Sie solche Literale in eine Zeile, ist der Schaden behoben.

Kapitel 2: Tutorium

2.1 Erste Sitzung: Anatomie eines C-Programms

Die Themen dieser Sitzung:

- Wie ist ein C-Programm aufgebaut?
- Elemente der Sprache C
- Funktionen
- Variablen

Information

Was halten Sie von einem Fahrlehrer, der einem Anfänger folgendes sagt? »Treten Sie auf das Pedal links unten, schieben Sie die Stange nach vorne links, lassen Sie das linke Pedal los, und treten Sie gleichzeitig auf das rechte.«. Wahrscheinlich halten Sie es für besser, erst einmal zu erklären, was Kupplung, Schaltung und Gaspedal sollen. Genau das passiert in dieser Sitzung, zum Schluß fahren wir auch ein Stück. Leider hat die Sprache C aber viel mehr Pedale, Hebel und Schalter als ein Auto, weshalb diese erste Fahrstunde etwas länger dauert.

Struktur eines C-Programms

Die Struktur eines C-Programms ist sehr einfach. Nach einem nicht unbedingt erforderlichen Einleitungsteil folgen Funktionen, Funktionen und nochmals Funktionen. Das ganze Programm und die Funktionen selbst müssen nach einem vorgeschriebenen Schema aufgebaut – man sagt auch strukturiert – sein. Am einfachsten läßt sich das System anhand eines Beispiels erklären. Wenden wir uns also dem Listing 2.1.1 zu.

```
/* Hiermit beginnt ein Kommentar,
   der erst endet, wenn "Stern-Strich folgt. */

#include <stdio.h> /* Ein Header-File einlesen  */
int i;             /* Eine Variable deklarieren */
hallo() /* Das ist eine Funktion */
{
   printf("Hallo, lieber Leser! \n");
}
/* ---------------------------------
   Hier steht häufig die Beschreibung
   der dann folgenden Funktion
   ---------------------------------
*/
int multipliziere(int mp, int mk)
{
  int ergebnis;        /* Eine lokale Variable */
  ergebnis = mp * mk;
  return(ergebnis);
}
main() /* Hier startet das Programm */
{
  hallo();          /* Funktion wird aufgerufen */
  i = 4711;         /* Variable erhält Wert     */
  printf("%d \n", i); /* Variable wird ausgegeben */
  i = multipliziere(3,4);
  printf("%d \n", i);
  printf("%d \n", multipliziere(3,4) );
}
```

Listing 2.1.1: Das erste Programm enthält schon fast alle typischen Elemente der Sprache C

Funktionen

Eine Funktion ist ein Stück Programm, das ausgeführt wird, wenn man die Funktion aufruft, d.h., den Funktionsnamen hinschreibt. Im Listing 2.1.1 ist zum Beispiel *hallo()* eine Funktion. Alle C-Befehle und auch die Funktionsaufrufe können nur innerhalb von Funktionen auftauchen.

☞ Im Gegensatz zu Pascal sind in C Schachtelungen verboten, d.h., innerhalb einer Funktion darf keine weitere Funktion definiert werden.

Jedes C-Programm hat mindestens eine Funktion namens *main()*. In dem kleinen Programm laut Listing 2.1.1 treten auch schon fast alle weiteren Struktur-Elemente der Sprache C auf. Nehmen wir uns diese Elemente doch einfach einmal der Reihe nach vor.

/ Das ist ein Kommentar */*

Kommentare sind erklärende Texte, die im fertigen Programm nicht mehr auftauchen, weil sie der Compiler einfach ignoriert. Sie sparen also nichts, wenn Sie die Kommentare weglassen – noch nicht einmal der Compilerlauf wird dadurch merklich schneller –, doch Sie verlieren viel, wenn Sie darauf verzichten. Auch Profis wissen nach ein paar Monaten nicht mehr, was sie da einst getrickst hatten, wenn sie nur den nackten Quelltext lesen. Der Sinn einer Funktion, ihre Abhängigkeiten, die Bedeutung von Datenstrukturen und vieles mehr bleiben ohne Kommentare unklar. Ein Kommentar beginnt mit /* und endet mit */. Verschachtelungen sind verboten. Folglich geht das nicht:

```
/* Im Kommentar /* darf nicht */ noch einer sein */
```

⚠ **In einem Kommentar darf die Zeichenfolge /* stehen, und das ist die Quelle für einen beliebten Fehler. Wenn Sie zum Beispiel im Listing 2.1.1 beim Kommentar »/* Variable erhält Wert */« den Abschluß (*/) weglassen, wird das folgende *printf()* nicht mehr ausgeführt, weil erst danach der immer noch offene Kommentar geschlossen wird.**

Direktiven

Ziemlich zu Beginn von Listing 2.1.1 finden Sie die Direktive *#include <stdio.h>*. Alle Direktiven beginnen mit dem #-Zeichen. Eine Direktive ist eine Anweisung an den Compiler. *#include* bedeutet, daß eine Textdatei eingeschlossen (eingelesen) werden soll. Texte, die immer wieder gebraucht werden, tippt man nicht jedesmal neu, sondern speichert sie auf der Disk und »included« sie dann. Die Wirkung ist die gleiche, als ob

der Text ab dieser Zeile eingetippt worden wäre. Die Datei *stdio.h* wird bereits mitgeliefert. Von diesen Header-Dateien – kurz H-Files genannt – gibt es sehr viele. Meistens definieren sie Konstanten mittels der Direktive *#define*. Die Direktive

```
#define ITEMTEXT 2
```

gibt der Konstanten »2« den symbolischen Namen ITEMTEXT. Ist das einmal definiert, können Sie anstatt 2 auch ITEMTEXT schreiben. Wenn solche Begriffe urplötzlich in einem Listing auftauchen, stammen sie immer aus einem Include-File. Unser ITEMTEXT ist zum Beispiel definiert, wenn Sie *#include <intuition.h>* schreiben, weil in diesem H-File auch *#define* ITEMTEXT 2 steht.

Befehle

Ein Programm ist eine Folge von Befehlen. In diesem Sinne sind auch Funktionsaufrufe Befehle. Ein einfacher Befehl ist zum Beispiel die Zuweisung i = 4711;. Nach Befehlen muß ein Semikolon folgen. Ansonsten – und deshalb – ist C formatfrei. Leerstellen, Tabulatoren und Zeilenschaltungen werden vom Compiler ignoriert.

☞ Beachten Sie jedoch, daß Stringliterale (Texte in Anführungszeichen) nicht in der nächsten Zeile fortgesetzt werden dürfen. Siehe auch die Hinweise am Ende der Einführung.

Schlüsselwörter und Namen

Schlüsselwörter sind Elemente der Sprache C. Die Tabelle 2.1.1 enthält nur die Schlüsselwörter des Standard-C. Einzelne Compiler können weitere Schlüsselwörter haben. Wichtig zu wissen ist, daß Schlüsselwörter nicht als Namen eingesetzt werden dürfen. Namen vergeben Sie selbst oder laden sie mit Include-Files. Im Listing 2.1.1 wird mit *int i;* eine Variable mit dem Namen *i* deklariert. Erlaubt sind alle Buchstaben, Ziffern und der Unterstrich.

Befehle

break	else	typedef
case	for	switch
continue	goto	while
default	if	
do	return	

Typen und Attribute

auto	float	static
char	int	register
const	long	union
double	short	volatile
enum	signed	void
extern	size of	

Reservierte Wörter

argc	envp
argv	main

Tabelle 2.1.1: Das ist das ganze Vokabular der Sprache C

☞ C unterscheidet streng zwischen Groß- und Kleinschreibung. *Anton*, *AnTon* und *anton* sind drei verschiedene Namen. Wenn Sie IF oder If anstatt *if* schreiben, meldet der Compiler einen Fehler.

Variablen und Datentypen

Eine Variable ist nichts weiter als ein symbolischer Name für einen Speicherbereich. Wir sagen nicht »lege die Zahl 4711 im Speicher ab Adresse 12345 ab«, sondern geben ihr einen Namen. Der Compiler weist dann der Variablen einen Speicherbereich zu und merkt sich, daß zum Beispiel die Variable *anton* die Adresse 12345 hat. Nun können aber unterschiedliche Variable durchaus verschieden viel Speicher belegen, und ob wir unter *anton* zwei oder acht Bytes ablegen wollen, muß der Compiler schon wissen. Diese Information erhält er über den Datentyp, kurz Typ genannt. Deshalb wird eine Variable immer deklariert als *Typ Name;*. Mit

```
int i;
```

wird eine Variable vom Typ *int* (Integer = Ganzzahl) angelegt, man sagt auch »deklariert«. Es gibt verschiedene vordefinierte Typen wie *int*, *float* (Fließkommazahl) oder *char* (Zeichen). Man kann aber auch eigene Typen definieren, was wir noch ausführlich kennenlernen werden. Eine Variable heißt Variable, weil man ihr immer wieder neue Werte zuweisen kann. Jede Variable sollte aber initialisiert werden, sprich, einen Anfangs-

wert bekommen. Das kann durch eine Zuweisung geschehen, zum Beispiel mit i = 4711; oder gleich bei der Deklaration mit

```
int i = 4711;
```

Sichtbarkeit von Variablen

Eine Variable kann zu Programmbeginn und damit außerhalb aller Funktionen deklariert werden. In diesem Fall steht sie allen Funktionen zur Verfügung, man nennt sie global oder extern. Im Listing 2.1.1 ist *i* eine globale Variable, *ergebnis* hingegen ist lokal für die Funktion *multipliziere()*. Lokale Variablen können nur innerhalb ihrer Funktion genutzt werden, alle anderen Funktionen kennen diese Variablen nicht.

Praxis

Wir wollen uns nun Listing 2.1.1 genauer ansehen und dabei besonders auf die vielen Feinheiten achten, auf die es in der Praxis ankommt.

Funktion schreiben und anwenden

Wie schon gesagt, besteht ein C-Programm hauptsächlich aus Funktionen, um diese müssen wir uns also besonders kümmern. Jede Funktion hat im Minimum die Form

```
NameDerFunktion()
{
}
```

Innerhalb der geschweiften Klammern stehen die Befehle, die beim Aufruf der Funktion ausgeführt werden sollen. Das nennt man den Funktionskörper, während die Zeile darüber Funktionskopf heißt. Bleibt der Raum zwischen den geschweiften Klammern leer, tut die Funktion gar nichts.

Die Main-Funktion

Jedes C-Programm hat eine Funktion namens *main()*. Der erste Befehl in *main()* ist ist auch derjenige, mit dem das Programm startet. Man spricht auch vom Startpunkt oder Entry-Point.

⚠ **Jede Funktion darf jede aufrufen, auch sich selbst (Rekursion). Sie dürfen jedoch niemals *main()* aufrufen, da dann das Programm in eine Endlosschleife geht.**

Das kürzeste C-Programm, das Sie schreiben können, sieht so aus:

```
main()
{
}
```

Sie können dieses Programm durchaus kompilieren, linken und starten. Es tut gar nichts, das aber blitzschnell. Da Zeilenschaltungen nur der Lesbarkeit des Textes dienen, können Sie das Programm auch schreiben als

```
main() {}
```

Funktionen aufrufen

Im Listing 2.1.1 gibt es außer *main()* noch die Funktionen *hallo()* und *multipliziere()*. Da *hallo()* keine Argumente hat (in den Klammern steht nichts), reicht für den Aufruf die Nennung des Namens. Der Aufruf *hallo();* sorgt dafür, daß die Befehle der Funktion ausgeführt werden.

⚠ **Auch wenn die Funktion keine Argumente braucht, dürfen Sie die runden Klammern beim Aufruf nicht weglassen. Tun Sie das, meldet der Compiler keinen Fehler, die Funktion wird aber auch nicht ausgeführt.**

Wie gleich noch gezeigt wird, können Funktionen auch einen Wert zurückgeben. Daraus resultieren verschiedene Formen des Aufrufs. In der Form

```
i = multipliziere(3,4);
```

wird der Rückgabewert einer Variablen zugewiesen. In diesem Beispiel hat dann *i* den Wert 12. Das entspricht der mathematischen Schreibweise, wo man zum Beispiel die Sinus-Funktion mit y=sin(x) notiert. Wie in der Mathematik kann die Funktion aber auch Teil eines Ausdrucks sein. 3 * sin(x) + 100 wäre ein Beispiel. In diesem Fall führt C die Funktion aus und setzt ihren Rückgabewert in den Ausdruck ein. Eine Funktion kann aber auch Argument einer anderen Funktion sein. Listing 2.1.1 zeigt diese Möglichkeit hiermit auf:

```
printf("%d \n", multipliziere(3,4) );
```

Die Funktion *printf()* wird weiter unten erklärt. Merken Sie sich an dieser Stelle: Egal, wo der Funktionsname auftaucht, die Funktion wird immer zuerst ausgeführt, danach wird der Rückgabewert irgendwo eingesetzt.

Argumente übergeben

Eine Funktion kann Argumente übergeben. Im Aufruf *multipliziere(3,4)* sind 3 und 4 Argumente. Die Funktion *multipliziere()* braucht zwei Argumente, nämlich die beiden Werte, die miteinander multipliziert werden sollen. Diese Argumente können wie hier Konstante sein, aber auch Variablen und sogar Funktionen sind zulässig.

Argumente kontra Parameter

Natürlich muß die Funktion mit den übergebenen Argumenten etwas anfangen können. Daher muß sie wissen, was die Argumente bedeuten, sprich, den Typ kennen. Dazu gibt es im Funktionskopf von *multipliziere()* (hier folgt er)

```
int multipliziere(int mp, int mk)
```

die beiden Platzhalter *mp* und *mk*. Beide sind vom Typ *int*. Solche Platzhalter nennt man Parameter. Erst, wenn die Funktion aufgerufen wird, werden Werte auf diese Plätze kopiert. Beachten Sie den feinen Unterschied: Die Funktion selbst hat Parameter. Wenn sie aufgerufen wird, werden für diese Parameter Argumente übergeben. Anders ausgedrückt: *mp* und *mk* sind Parameter, im Aufruf *multipliziere(3,4)* sind 3 und 4 Argumente. Weil das so oft verwechselt wird, eine kleine Eselsbrücke: Nur der Aufrufer kann argumentieren, die Funktion selbst kann den Auftrag nur ausführen.

Rückgabe eines Wertes

Wenn eine Funktion einen Wert zurückgeben soll, muß der Typ des Rückgabewertes – Return-Wert genannt – im Funktionskopf vor dem Namen stehen. Wie Sie sehen, heißt es »int multipliziere(int mp, int mk)«, aber nur *hallo()*.

Das verwirrende an der Geschichte ist, daß eine Funktion auch dann einen Wert zurückgibt, wenn man den Return-Typ wie bei *hallo()* gar nicht definiert hat. In diesem Fall ist der Rückgabewert vom Typ *int*, aber ziemlich sinnlos, da zufällig. Es ist aber erlaubt, mit zum Beispiel i = hallo(); diesen Wert anzufordern. Andererseits können Sie einen definierten Return-Wert auch einfach ignorieren. So dürfen Sie anstatt *i = multipliziere(3,4);* auch einfach *multipliziere(3,4);* schreiben. Auch das bringt nichts, weil das Rechenergebnis dann im Leeren landet.

Um hier Klarheit zu schaffen, gibt es den Typ *void* mit der Bedeutung »der Return-Wert wird nicht gebraucht«. Deshalb schreibt man die Funktion *hallo()* korrekter so:

```
void hallo() /* Das ist eine Funktion */
{ printf("Hallo, lieber Leser! \n");
}
```

Jetzt weiß der Compiler, was Sie meinen, und wird deshalb einen Zugriff auf den Return-Wert anmahnen. Wenn die Funktion *hallo()* als *void* deklariert ist, wird ein Befehl der Art *i = hallo();* vom Aztec C als ungültige Typ-Konversion »angemeckert«, Lattice C beklagt einen ungültigen Void-Operanden.

Damit die Funktion einen Wert zurückgeben kann, muß man allerdings noch etwas tun. Die letzte Zeile der Funktion lautet:

```
return(ergebnis);
```

Der Return-Befehl benötigt als Argument einen Wert, der wie hier eine Variable sein kann, aber auch eine Konstante oder ein Ausdruck ist erlaubt. Mit einem Ausdruck können wir die Funktion

```
int multipliziere(int mp, int mk)
{
  int ergebnis;        /* Eine lokale Variable */
  ergebnis = mp * mk;
  return(ergebnis);
}
```

kürzen auf

```
int multipliziere(int mp, int mk)
{
  return(mp * mk);
}
```

Die im Kopf deklarierten Variablen *mp* und *mk* können wie lokale Variablen behandelt werden, was beim nächsten Punkt noch deutlicher wird.

ANSI kontra K&R

Die Deklaration der Art

```
int multipliziere(int mp, int mk)
```

ist erst in den 5er-Versionen von Aztec C bzw. Lattice C erlaubt. Sie folgt dem neusten ANSI-Standard (einer anerkannten amerikanischen Norm). Mit den älteren Compilern können Sie nur die folgende Form wählen, die neueren Versionen akzeptieren sie auch. Diese Form entspricht dem sogenannten K&R-Standard, so benannt nach den Erfindern von C, die Kernighan und Ritchie heißen.

```
int multipliziere(mp, mk)
int mp;
int mk;
{
  return(mp * mk);
}
```

Das kann man noch leicht kürzen, weil Deklarationen desselben Typs auch aufgezählt werden dürfen, was dann so aussieht:

```
int multipliziere(mp, mk)
int mp, mk;
{
  return(mp * mk);
}
```

Damit ist der Unterschied rein textuell betrachtet nicht mehr sehr groß. Sie haben die freie Auswahl.

Prototypen

Ein anderer Punkt des ANSI-Standards bringt allerdings einen erheblichen Unterschied. Schreiben Sie nämlich vor der Funktionsdefinition – praktisch zu Programmbeginn – einen sogenannten Prototyp, kann der Compiler prüfen, ob Sie der Funktion auch die richtigen Argumenttypen übergeben. Dazu reicht es, eine Zeile an den Programmanfang zu setzen, zum Beispiel diese:

```
int multipliziere(int mp, int mk);
```

Praktisch ist das die Kopfzeile der ANSI-Form-Funktion. Doch beachten Sie, daß jetzt ein Semikolon am Ende stehen muß. Die Variablennamen (*mp* und *mk*) können dieselben, aber auch andere sein, Hauptsache die Typen stimmen.

☞ Obwohl der Prototyp wie die Kopfzeile der ANSI-Form-Funktion aussieht, ist das Prototyping auch möglich, wenn die Funktionen nach dem alten K&R-Standard definiert werden.

Wenn Sie jetzt die Funktion falsch aufrufen, zum Beispiel mit *multipliziere(3, 4.77)* (das zweite Argument ist keine Ganz- sondern eine Fließkommazahl), wird der Compiler mit »argument type mismatch« eine Warnmeldung ausgeben. Lassen Sie das Prototyping weg, erscheint das falsche Ergebnis 12, weil der Compiler einfach von der 4.77 nur den Ganzzahlanteil (4) übernimmt. Der korrekte Prototyp für eine Funktion ohne Return-Wert und ohne Parameter wie unser *hallo()* lautet:

```
void hallo(void);
```

Wichtig ist das »Prototyping« bei Funktionen, die Sie nicht so genau kennen, weil Sie diese nicht selbst geschrieben haben. Solche Funktionen – nämlich die »eingebauten« Amiga-Funktionen – werden Sie später sehr häufig nutzen. Die dafür erforderlichen Prototypen stehen bei Aztec C in der Datei *functions.h*. Es empfiehlt sich dann, ziemlich zu Programmbeginn zu schreiben:

```
#include <functions.h>
```

☞ Bei Aztec C reicht es nicht, die Prototypen zu definieren oder als Include-File zu laden. Zusätzlich müssen Sie die Ausgabe der Warnmeldung mit der Compiler-Option *-wa* einschalten. Zum Beispiel müssen Sie den Quelltext *hallo.c* mit *cc -wa hallo* kompilieren.

Die Funktion printf()

Die Sprache C kennt natürlich einige (wenige) Befehle oder Schlüsselworte, doch darunter werden Sie nichts finden, was für die Ein- oder Ausgabe – englisch I/O (Input/Output) – geeignet ist. Einen Befehl wie »PRINT« in Basic oder »write« in Pascal kennt C nicht. Statt dessen werden mit dem Compiler Bibliotheken (Libraries) geliefert, welche die erforderlichen Funktionen bieten. Eine Library ist nichts weiter als eine bereits kompilierte Sammlung von Funktionen. Mittels des Linkers werden diese Funktionen Ihrem Programm hinzugefügt.

Die wohl bekannteste Funktion aus dieser Sammlung heißt *printf()*. In Listing 2.1.1 setzen wir *printf()* für eine simple Aufgabe ein, nämlich die Ausgabe des Inhalts der Variablen *i*, eine schlichte »12« soll auf dem Schirm erscheinen.

```
printf("%d \n", i);
```

Etwas kompliziert wird die Sache, weil »printf« das Kürzel für »print formatiert« ist. In Basic wäre das PRINT USING. Deshalb brauchen wir immer einen Format-String, das sind diese Kürzel in den Anführungszeichen. Dabei sind drei Dinge zu unterscheiden:

- Format-Code
- Escape-Zeichen
- Reiner Text

Der Format-Code beginnt immer mit dem %-Zeichen, dem ein Buchstabe folgt. Hierfür einige Beispiele:

d	(oder i) Dezimalzahl mit Vorzeichen
u	Dezimalzahl ohne Vorzeichen
x	(oder X) Hexadezimalzahl
f	Fließkommazahl
c	Zeichen
s	String

Das schauen wir uns gleich mal in der Praxis an.

```
main()
{
  int i = 4711;
  float f = 47.13;
  printf("%d %X %7.2f \n", i, -1, f);
}
```

Listing 2.1.2: printf() in der Praxis

Listing 2.1.2 ist ein lauffähiges C-Programm, mit dem Sie später noch weiterarbeiten werden. Beachten Sie zuerst, daß wir hier zwei Variable, nämlich *i* und *f* deklariert und auch gleich initialisiert (mit einem Startwert versehen) haben. Mehr dazu im nächsten Abschnitt. Beide Variablen sind lokal für die Funktion *main()*, können also in anderen Funktionen nicht genutzt werden. Doch nun zu *printf()*. Das Ergebnis sieht so aus:

```
4711  FFFFFFFF  47.13
```

Da wir drei Zahlen ausgeben wollen, brauchen wir auch drei Format-Codes. *%d* gilt für die Ganzzahl 4711, *%X* stellt –1 als Hexadezimalzahl dar (ergibt FFFFFFFF). Beachten Sie, daß auch Konstanten ausgegeben

werden können (hier –1) und – viel wichtiger – *printf()* auch zwischen den Zahlenbasen automatisch konvertiert.

Nun zu den »Kommazahlen«: Nur *%f* schreibt man eigentlich nie. Statt dessen steht hier *%7.2f* mit der Bedeutung, daß die Fließkommazahl sieben Schreibstellen belegen und zwei Nachkommastellen haben soll.

Bliebe noch das Escape-Zeichen, das immer mit einem Backslash (\) eingeleitet wird. Diese Zeichen können an beliebiger Stelle im Format-String auftauchen. Hier einige davon:

\a Alarm (gibt einen Beep aus)
\b Backspace (Rückschritt)
\n Neue Zeile
\t Tabulator

Jetzt wissen Sie auch, warum ich des öfteren *\n* eingesetzt habe. Das ist immer nötig, wenn nach der Ausgabe eine neue Zeile begonnen werden soll. Auch bei nur einem *printf()* sollten Sie daran denken, andernfalls steht nach dem Programmende der Cursor im CLI sonstwo.

Typen, Variablen und Konstanten einsetzen

Für die Amiga-Programmierung werden wir hauptsächlich die »eingebauten« ROM-Funktionen nutzen. Diese Funktionen benötigen Argumente in Form von Variablen und Konstanten und geben Werte zurück, für die Variable bereitgestellt werden müssen. Wichtig dabei ist, daß all diese Parameter vom richtigen Typ sind. Wenn Sie die Typ-Problematik beherrschen, ist der Rest ganz einfach, also packen wir's an.

Es gibt einfache Variable, aber namentlich keine schwierigen. Ob die anderen – die strukturierten Variablen – tatsächlich schwieriger sind, werden Sie nach der dritten Sitzung beurteilen können. Jetzt geht es den einfachen Variablen an den Kragen. Bei allen Variablen müssen Sie immer drei Punkte im Auge haben:

- Deklarieren
- Initialisieren
- Zuweisen

Jede Variable muß zuerst deklariert werden. Dann muß die Variable initialisiert werden, sprich, einen Wert erhalten. Das kann gleich mit der Deklaration geschehen oder später durch eine Zuweisung. Das Prinzip

haben Sie schon im Informationsteil kennengelernt, hier einige praktische Beispiele:

```
char zeichen = 'a';
int ganzzahl = -12345;
long lange_ganzzahl = 2147483647;
float kommazahl = 11.1;
double doppeltgenaue_kommazahl = 66.123456789;
int i,j,k,l,m; /* bevorzugte Namen für "int's" */
i = 11;
j = i; /* schneller als "j = 11" */
zeichen = 'b';
ganzzahl = 4711;
```

Wie Sie schon erkannt haben, können die unterschiedlichen Typen verschieden große Zahlen aufnehmen. Im Standard-C sind die Typen laut Tabelle 2.1.2 definiert. Daß die Amiga-Compiler darüber hinaus weitere Typen kennen, werden wir noch lernen.

Typ-Name	Alternativ-Namen	Wertbereich
char	signed char	-128...127
int	signed, signed int	-32768...32767
short	short int, signed short, signed short int	-32768...32767
long	long int, signed long, signed long int	-2147483648 bis 2147483647
unsigned char	keine	0...255
unsigned	unsigned int	0...65,535
unsigned short	unsigned short int	0...65535
unsigned long	unsigned long int	0...4294967295
enum	keine	-32768...32767
float	keine	3.4E æ 38 (7 Digits)
double	keine	1.7E æ 308 (15 Digits)
long double	keine	1.2E æ 4932 (19 Digits)

Tabelle 2.1.2: Diese Typen bietet Standard-C

Bei der Vergabe von Namen sind einige Regeln zu beachten:

- Erlaubt sind Buchstaben, Ziffern und der Unterstrich.
- Das erste Zeichen darf keine Ziffer sein. Ein Unterstrich als erstes Zeichen ist zwar erlaubt, doch per Konvention sind solche Namen für den Compiler reserviert.
- Alle Schlüsselwörter (siehe Tabelle im Informationsteil) sind verboten.
- Variablennamen werden in aller Regel kleingeschrieben, Konstanten hingegen groß.
- C unterscheidet streng Groß- und Kleinschreibung. *Anton*, *anton* und *anTon* sind drei verschiedene Namen.
- Nur die ersten 31 Zeichen eines Namens sind signifikant. Sie können zwar längere Namen vergeben, doch werden zwei Namen nicht unterschieden, wenn die ersten 31 Zeichen gleich sind.

Bei Zuweisungen wie *ganzzahl = 12345;* ist die Zahl 12345 eine Konstante. Aber auch in *zeichen = 'a'* ist *'a'* eine Konstante.

⚠ **Beachten Sie, daß die Hochstriche beim Amiga über die Tastenkombination Alt+ä erzeugt werden müssen.**

Es gibt also offensichtlich auch bei den Konstanten verschiedene Typen, und die wollen wir uns einmal ansehen.

Konstante	Typ
255	int (dezimal)
0xFF	int (hexadezimal für 255)
0377	int (oktal für 255)
255L	long int
255U	unsigned int
0xFFul	long unsigned int
1.77	float
.123	float
13.7E2	float
"Hallo"	String-Konstante (siehe Sitzung 3)

Tabelle 2.1.3: Die verschiedenen Typen bei Konstanten

Für uns – bzw. die Amiga-Programmierung – ist besonders das Suffix »L« wichtig. Wenn Sie nichts anderes spezifizieren und die Zahl im Bereich von –32768...+32767 liegt, nimmt C den Typ *int* an. Einige Amiga-Funktionen erwarten aber auch, daß kleine Zahlen vom Typ *long* sind. Daher sieht man häufig Konstanten wie *1L* oder *0L*.

⚠ **Eine führende Null heißt »oktal« (Zahlenbasis 8). Folglich hat 0377 den Wert »3 * 64 + 7 * 8 + 7 = 255«**

Wir bringen die Programme zum Laufen

Die beiden Listings 2.1.1 und 2.1.2 sind Programme, oder? Nun, wie sie da so stehen, sind sie reiner Text und sonst gar nichts. »Programmieren« ist ein Sammelbegriff für vier Aktivitäten, nämlich:

- Editieren
- Kompilieren
- Linken
- Testen

Editieren heißt »Eingabe des Textes« und dann später – und viel häufiger – »Ändern des Textes«, um Fehler zu beseitigen oder um das Programm zu ändern. Danach folgt das Kompilieren, also die Übersetzung des Textes in die Maschinensprache. Dadurch entsteht ein sogenanntes Objekt-File. Das ist aber noch nicht lauffähig.

Der Linker hat viele Aufgaben. Zuerst muß er alle Library-Funktionen, die Sie ihm nennen, in das Objekt-File einfügen. Dann muß er den sogenannten Startup-Code hinzubinden. Ein Amiga-Programm kann nämlich nicht einfach loslaufen, sondern hat einige Spielregeln zu beachten. Schließlich muß der Code noch mit einem sogenannten Header versehen werden, an dem Amiga-DOS erkennt, daß es sich um ein ausführbares Programm handelt.

War der Linker erfolgreich, können Sie das Programm starten und testen. Für den Start reicht vorerst die Eingabe des Namens im CLI bzw. in der Shell. Später lernen wir, wie man ein Programm mit einem Icon versieht, so daß es auch von der Workbench aus gestartet werden kann. Meistens stellt man im Test Fehler fest, woraufhin dann die Folge »Editieren, Kompilieren, Linken, Testen« wieder zu durchlaufen ist. Das nennen die Programmierer »Strafschleife«, die dafür benötigte Zeit heißt »Turn-Around-Zeit«.

Anfangs ist die Strafschleife etwas kürzer, weil viele Fehler schon der Compiler findet. Logische Fehler kann er allerdings nicht erkennen.

☞ Praktisch sind die vier Schritte recht einfach auszuführen. Wir setzen voraus, daß Sie Ihr System entsprechend den Vorschlägen in Kapitel 1 eingerichtet haben und daß der dort vorgeschlagene Test erfolgreich bestanden wurde. Im Kapitel 1.6.4 finden Sie auch Vorschläge für den Fall, daß etwas nicht funktionieren sollte.

Als Festplattenbesitzer wechseln Sie in das Directory, in dem Sie Ihren Compiler (nebst Zubehör) abgelegt haben. Je nach Installation müssen Sie noch *execute az* oder *execute lat* eingeben (siehe auch Kapitel 1.5 und 1.6). In den Diskettenversionen starten Sie Ihren Amiga mit den eingelegten C-Disketten.

Nun geben Sie Listing 2.1.1 mit einem Editor ein und speichern den Text als *hallo.c*. Jeder andere Name ist auch gut, nur muß er immer den Extender *.c* (Punkt c) haben.

☞ Die folgenden Compile-/Link-Beispiele gelten nur für Listing 2.1.1. Beachten Sie den untenstehen Abschnitt »Linken mit *float*«.

Unter Aztec C geben Sie dann ein:

```
cc hallo [Return]
ln hallo -lc [Return]
```

Unter Lattice C geben Sie ein

```
lc -L hallo [Return] (-L groß schreiben!)
```

Die Compiler und Linker geben allerhand Text aus. Solange da nichts von »Error« steht, ist alles OK.

☞ Tips zur Fehlersuche finden Sie im Kapitel 1.6.4.

Sie können nun das Programm durch die Eingabe des Namens (hallo) starten. Unter Lattice C werden Sie die Warnung »function returns value mismatch« sehen. Wenn Sie das stört, schreiben Sie nicht *main()*, sondern *void main()*.

Linken mit float

Sobald Sie Fließkommazahlen anziehen, müssen Sie mit der m(Mathematik)-Library linken. Wenn Sie beispielsweise Listing 2.1.2 unter dem Namen *fkz.c* gespeichert hatten, gilt:

Unter Aztec C:

```
cc hallo [Return]
ln hallo -lm -lc [Return]
```

Unter Lattice C:

```
lc -Lm hallo [Return]
```

Workshop

Checkliste

1. Wie wird der Typ des Rückgabewerts einer Funktion bestimmt?
2. Warum müssen Variable deklariert werden?
3. Was ist der Unterschied zwischen externen und lokalen Variablen?

Ideen für eigene Übungen

1. Im Listing 2.1.2 wurde die Anwendung von *printf()* demonstriert. Schreiben Sie auf dieser Basis ein Programm, das eine formatierte Tabelle ausgibt, zum Beispiel diese:

```
Titel 1     Titel 2      Titel 3
-----------------------------
   47         17.5        22.123
   66        134.7         1.007
 1234          1.2       123.456
```

2. In die Variablen *mp, mk* und *ergebnis* sollen die Zahlen 3, 4 und 12 gebracht werden. Dann soll *printf()* unter Anziehung der Variablen den Text »3 mal 4 ist 12« ausgeben.

   Zwei Tips dazu:

   In *printf()* wird jede Art von Text im Format-String direkt ausgegeben. Auch eine Form wie »printf("Titel 1 Titel 2 Titel 3 \n ");« ist zulässig.

   Sie können im Format-String Texte auch zwischen die Format-Codes setzen.

2.2 Zweite Sitzung: C-Spezialitäten für die Amiga-Programmierung

Die Themen dieser Sitzung:

- Was sind Datenstrukturen?
- Arrays
- C-String
- Zeiger
- C-struct
- Typ-Casting
- #include

Information

Ein Amiga-typisches Programm mit seinen Windows und Menüs entsteht im Prinzip ganz einfach. Man lädt Datenstrukturen mit den richtigen Werten und ruft dann die schon im ROM vorhandenen Funktionen auf. Leider sind diese Datenstrukturen sehr komplexe Gebilde und schwer begreifbar, wenn man sie das erste Mal sieht. Deshalb lernen Sie in dieser Sitzung anhand einfacher Beispiele, was Datenstrukturen sind und wie man mit ihnen umgeht. Noch einige C-Leckerbissen, die in diesem Zusammenhang vorkommen, nehmen wir uns gleich mit vor.

Einfache und strukturierte Variablen

Einfache Variablen haben wir in der zweiten Sitzung schon kennengelernt. Typisch für diese ist, daß sie immer nur genau ein Datum aufnehmen können. Wie unpraktisch das manchmal ist, zeigt das Beispiel der Zeichenvariablen. Es macht sicherlich keinen Spaß, wenn man das Wort »Hallo« so speichern müßte:

```
char c1 = 'H';
char c2 = 'a';
char c3 = 'l';
char c4 = 'l';
char c5 = 'o';
```

Tatsächlich schreibt man dafür

```
char c[] = "Hallo";
```

Arrays

Die rechteckigen Klammern hinter einem Variablennamen kennzeichnen einen Array. Das ist eine Sammlung von Daten des *gleichen* Typs unter einem Namen. Das obige Beispiel ist ein Zeichen-Array, genauso sind Zahlen-Arrays möglich. Einen Array zu deklarieren, ist recht einfach, wie das folgende Beispiel zeigt:

```
main()
{ int ar[3];
  ar[0] = 10;
  ar[1] = 20;
  ar[2] = 30;
  printf("%d %d %d \n", ar[0], ar[1], ar[2] );
}
```

In diesem Beispiel ist *ar* ein Array vom Typ *int*, kurz int-Array genannt. Dieser Array hat drei Elemente, die über den *Index* 0, 1 oder 2 angesprochen werden.

☞ Beachten Sie, daß in C das erste Element immer den Index 0 hat im Gegensatz zu Pascal oder Basic (mit OPTION BASE 1). Da dieser Array drei Elemente hat (*ar[0], ar[1]* und *ar[2]*) ist *ar[3]* nicht definiert.

Sie können ein einzelnes Array-Element wie eine einfache Variable des gleichen Typs auffassen, weshalb eine Zuweisung der Art

```
einf_var = ar[2];
```

durchaus zulässig ist. Wichtig ist ein anderer Aspekt. Spätestens bei großen Arrays wird nämlich die Angabe des Index als Konstante ausgesprochen lästig. Meistens zusammen mit Schleifen (siehe vierte Sitzung) setzt man deshalb für den Index eine einfache Variable ein, wozu Sie wissen müssen:

```
ar[2] = 177; ist gleichwertig mit   i = 2;
                                    ar[i] = 177;
```

Niemand hindert Sie allerdings daran, der Indexvariablen einen größeren Wert als 2 zu geben. Ein sehr häufiger Fehler ist der Versuch, bei einem mit *int ar[3]* deklarierten Array das undefinierte Element *ar[3]* anzusprechen. Sie können damit sehr viel Schaden anrichten, und niemand warnt Sie!

⚠ **Im Gegensatz zu Basic oder Pascal prüft C nicht den Array-Index. Sie können bei einem mit int *ar[3]* deklarierten Array durchaus ar[1733] = 4711; schreiben. Damit können Daten in wichtigen Speicherbereichen überschrieben werden, Guru-Meldungen sind möglich.**

Die Besonderheiten eines C-Strings

Ein String in C – kurz C-String genannt – ist prinzipiell nur ein Array vom Typ *char*. Er wird zum String, wenn das letzte Zeichen ein Null-Byte ist, genauer: Der String endet, sobald in der Folge ein Null-Byte auftritt. Dieses Null-Byte ist nicht mit dem ASCII-Code des Zeichens »0« (Wert 48) zu verwechseln. Wenn Sie dem String mit

```
char c[] = "Hallo";
```

gleich bei der Deklaration einen Text zuweisen, setzt der Compiler automatisch das Null-Byte ein. Ferner wird er bei einer leeren Klammer selbst zählen und für die Dimension 5 einsetzen (vier Zeichen und das Null-Byte).

Beachten Sie aber, daß Sie auf diese Art einer Stringvariablen keinen Text zuweisen können. Das geht nur mitttels der Funktion *strcpy()* (String Copy), beispielsweise so:

```
void main()
{ char str[20];
  strcpy(str, "hallo");
  printf("%s \n", str);
}
```

Die Bedeutung von Zeigern in C

Es ist unmöglich, ein echtes Amiga-Programm ohne Zeiger zu schreiben, und überhaupt nutzt jedes bessere C-Programm Zeiger (englisch: Pointer) mehr oder weniger intensiv (meistens mehr). Man braucht Zeiger unter anderem für diese Operationen:

- Manipulation von Strings und Arrays
- Rückgabe von mehr als einem Wert bei Funktionen
- Zugriff auf Datenstrukturen
- Zugriff auf Speicherbereiche

Zeiger sind letztlich immer Adressen, und Adressen sind vom Typ *unsigned long*. Das gilt aber nur auf Maschinenebene. In C sind Zeiger an einen Typ gebunden. Mit der Deklaration

```
int *ptr;
```

wird ein Zeiger mit dem Namen *ptr* geschaffen, der nur auf ein Objekt vom Typ *int* zeigen kann. Die Typbindung ist deshalb nötig, weil C im Rahmen seiner Zeiger-Arithmetik (siehe Praxisteil) zum Beispiel von einem Array-Element auf das nächste schalten kann. Dazu muß aber der Wert des Zeigers (praktisch die Adresse) bei char-Arrays um 1 und bei long-Arrays um 4 erhöht werden. Daß man diese Typbindung mittels des Type-Casting (siehe Praxisteil) umgehen kann und manchmal sogar umgehen muß, ist ein anderes Problem.

Praxis

In den ersten beiden Kapiteln haben wir den Umgang mit den Compilern und Linkern gelernt. Ab jetzt gilt: Jedes Programmlisting mit einer main()-Funktion ist lauffähig, doch glauben Sie das nicht, probieren Sie es aus! Ernsthaft: Sie lernen C am besten, wenn Sie immer wieder Programme zum Laufen bringen. Sie werden dabei anfangs Fehler machen, aber auch das muß sein. Nur die selbst gemachten und ergründeten Fehler macht man (meistens) nie wieder.

Einen Array deklarieren und nutzen

Wissen Sie, was das sein soll?

```
00100
00100
11111
00100
00100
```

Es ist ein Kreuz aus Sicht des Computers. Für ihn ist ein Bildpunkt ein Bit, das nur 0 oder 1 sein kann. Jede Art von Grafik ist so ein Bit-Muster. 8 Bit sind bekanntlich 1 Byte, 2 Byte sind ein Wort oder aus unserer C-Sicht eine Zahl vom Typ *int*. Computer-intern wird jede Zahl als Bit-Muster gespeichert, zum Beispiel die Zahl 7 als 00000111. Folglich kann man ein Computer-Bild auch als eine Folge von Zahlen darstellen, und genau das tut der Amiga. Wenn Sie einen neuen Mauszeiger erzeugen wollen, müssen Sie der Funktion *SetPointer()* unter anderem einen Array übergeben, mit dessen Zahlen der Mauszeiger gemalt wird. Es gibt aber auch andere Funktionen, die Arrays brauchen. Zum Beispiel übergeben Sie der Funktion *PolyDraw()* einen Array, in dem die Koordinaten der Eckpunkte eines zu zeichnenden Vielecks stehen. Langer Rede kurzer Sinn: Für die Amiga-Programmierung brauchen Sie Kenntnisse über Arrays, also packen wir's an.

Betrachten wir noch einmal das Beispiel aus dem Informationsteil:

```
main()
{ int ar[3];
  ar[0] = 10;
  ar[1] = 20;
  ar[2] = 30;
  printf("%d %d %d \n", ar[0], ar[1], ar[2] );
}
```

☞ Wenn Ihr Compiler die folgenden Listings nicht akzeptiert, zum Beispiel Lattice Version 4.0, deklarieren Sie die Arrays global (vor *main()*).

```
int ar[3] = {10, 20, 30}; /* global */
main()
{  printf("%d %d %d \n", ar[0], ar[1], ar[2] );
}
```

So einem kleinen Array drei Zahlen einzeln zuzuweisen, geht ja noch, aber bei 20 Werten ist das schon lästig. Deshalb schreibt man das besser so:

```
main()
{ int ar[3] = {10, 20, 30}; /* lokal */
  printf("%d %d %d \n", ar[0], ar[1], ar[2] );
}
```

Wie einfache Variable kann man also auch Arrays gleich mit der Deklaration initialisieren. Die Dimension (die »3« in den rechteckigen Klammern) darf man dann auch weglassen.

☞ Im Gegensatz zu manchem Pascal, wo so etwas typisierte Konstante heißt, kann in C die Dimension größer sein als die Anzahl der aufgeführten Werte. Sie können aber auch in C nicht mehr Werte aufführen, als die Dimension erlaubt.

Arrays können beliebig viele Dimensionen haben, wenn es auch praktisch kaum mehr als zwei sind. Und so sieht das aus (die Erklärung von *short* kommt gleich):

```
main()
{ short int ar[3][4] =
      {
         {01, 02, 03, 04},
         {11, 12, 13, 14},
         {21, 22, 23, 24}
       };
  printf("%d %d \n", ar[0][0], ar[2][3] );
}
```

Bei einem zweidimensionalen Array definieren Sie erst Zeile 0, dann Zeile 1 usw. Jede Zeile setzen Sie in ihr eigenes Klammerpaar, dazwischen kommt je ein Komma.

Dieser Array hat drei Zeilen und vier Spalten. Die »01« und die »24« werden von *printf()* ausgedruckt. Beachten Sie auch hier, daß immer ab Null gezählt wird, weshalb *ar[2][3]* die »24« anspricht.

int und short int

Nun ändern Sie einmal die letzte Zeile des obigen Programms wie folgt:

```
printf("%d %d %d \n", ar[0], ar[1], ar[2] );
```

Sie werden drei Zahlen sehen, zum Beispiel diese:

```
12826592 12826600 12826608
```

Warum Sie damit tatsächlich die Adressen der drei Array-Zeilen drucken, werden Sie im Abschnitt über Zeiger noch lernen. Wichtig ist hier die Tatsache, daß die Differenz zwischen den Zahlen 8 ist. Jede Zeile hat vier Elemente, folglich belegt jedes Element 2 Byte. Das tut es aber nur, weil wir anstatt *int* hier *short int* geschrieben haben (nur *short* reicht auch). Der Typ *int* benötigt zwar nur 2 Byte, doch viele Amiga-Compiler legen ihn in 4 Byte ab. Nun macht es bei großen Arrays schon sehr viel aus, ob man 400 oder 800 Kbyte Speicher belegt, aber noch wichtiger ist folgendes: Die oben geschilderten Grafik-Funktionen, die in Arrays ihre »Bilder« erwarten, unterstellen das kurze int-Format.

Strings anlegen und damit umgehen

Es gibt viele Stringfunktionen, doch in der Amiga-Praxis kommen Sie mit sehr wenigen aus. Mit Listing 2.2.1 wollen wir das üben.

```
main()
{
  char str1[80], str2[80];
  strcpy(str1, "Guten Tag, mein lieber ");
  printf("Wie heißen Sie? ");
  scanf("%s", str2);
  strcat(str1,str2);
  printf("%s \n",str1);
}
```

Listing 2.2.1: Der Umgang mit Strings

Das Programm soll nach einem Namen fragen und nach dessen Eingabe »Guten Tag, mein lieber <Name>« ausgeben. Dazu deklariert es zwei Strings, nämlich *str1* und *str2*. Beide können maximal 79 Zeichen aufnehmen. Mit Hilfe der Funktion *strcpy()* (string copy) wird der Text »Guten Tag, mein lieber« nach *str1* kopiert.

Neu ist jetzt die Funktion *scanf()*. Praktisch ist das die Umkehr von *printf()*, sogar die Format-Codes sind die gleichen. In diesem Fall wartet *scanf()* wegen des Format-Codes *%s* auf die Eingabe eines Strings. Danach steht in *str2* ein Name.

Jetzt kommt die Funktion *strcat(str1,str2)* zum Einsatz. Sie verbindet *str1* mit *str2* und schreibt das Ergebnis nach *str1*.

⚠ **Der hier gezeigte Umgang mit *scanf()* trifft so nur auf Arrays zu. Bei einfachen Variablen muß vor deren Namen der Adreßoperator & stehen, wie das folgende Beispiel zeigt:**

```
main()
{ int zahl;
 printf("Gebe eine Zahl ein: ");
 scanf("%d", &zahl); /* Beachte das "&" ! */
  printf("%d\n", zahl);
}
```

Definieren eines Strukturtyps

Der Array ist bereits eine Datenstruktur, hat jedoch für manche Anwendungen einen Nachteil: Alle Elemente müssen vom selben Typ sein. Das ist oft sehr unpraktisch, denn häufig genug gehören Variable verschiedenen Typs logisch zusammen. Für diesen Zweck gibt es in C den struct-Typ. Nehmen wir das Beispiel einer Personal-Datei, so sieht das typisch so aus:

```
struct person
{
  char name[50];
  char strasse[30];
  int plz;
  char ort[20];
  float gehalt;
}
```

Damit haben wir einen neuen Datentyp namens *struct person* geschaffen. Das ist erst einmal nur ein Typ wie *int* oder *char* auch. Beachten Sie einen wichtigen Unterschied zu den vordefinierten Typen: Diese heißen kurz nur *int* oder *char*, unser neuer Typ heißt nicht *person*, sondern *struct person*. Das *struct* ist obligatorisch, bei dem Typnamen sind Sie frei.

Deklarieren einer Strukturvariablen

Wie üblich, kann man auch hier nicht direkt mit dem Typ arbeiten, sondern braucht eine Variable. Das geht ganz einfach so:

```
struct person meier;
```

Initialisieren von Strukturvariablen

Um eine Komponente dieser Strukturvariablen anzusprechen, braucht man nur den Namen, einen Punkt und den Bezeichner aus der Deklaration. Klingt schwierig, ist es aber nicht, wie dieses Beispiel zeigt:

```
meier.gehalt = 4200.50;
```

Strings behandelt man auch wie üblich, das heißt, man muß sie mit *strcpy()* laden. Das sollten Sie jetzt einmal üben. Geben Sie Listing 2.2.2 ein. Sie haben alles richtig gemacht, wenn das Programm »Franz Meier verdient 4200.50 DM« ausgibt.

☞ Wir haben eine float-Variable eingesetzt, weshalb wir bei Aztec mit *-lm -lc* und bei Lattice mit *-Lm* linken müssen.

```
struct person
{
  char name[50];
  char strasse[30];
  int plz;
  char ort[20];
  float gehalt;
}

main()
{  struct person meier;
   strcpy(meier.name, "Franz Meier");
   meier.plz = 8900;
   meier.gehalt = 4200.50;
   printf("%s verdient %7.2f DM\n", meier.name,
                                    meier.gehalt);
}
```

Listing 2.2.2: struct in der Praxis

Sie können nun zig Variable wie *meier*, *müller* und *schulze* vom Typ *struct person* deklarieren, doch Sie werden es wahrscheinlich als unpraktisch empfinden, auf diese Art eine Personaldatei oder eine Adreßverwaltung aufzubauen. Doch Sie müssen sich nur eines merken: Alles, was mit den vordefinierten Typen möglich ist, ist auch bei den selbstdefinierten erlaubt. Nach dieser allgemeinen Typregel können Sie auch einen Array wie diesen anlegen:

```
struct person datei[100];
```

Die Punktregel gilt immer noch, also können Sie schreiben:

```
i=13;
datei[i].gehalt=4200.50;
```

Diesen Weg sollten Sie einmal übungshalber verfolgen. Ich möchte Sie jedoch zuerst mit einer beim Amiga sehr häufig auftretenden Möglichkeit bekannt machen. So, wie wir schon einfache Variable und Arrays gleich bei der Deklaration initialisiert haben, muß das nach der allgemeinen Typregel auch beim *struct* funktionieren. Sie können deshalb in Listing 2.2.2 die main()-Funktion auch so schreiben:

```
struct person
{
  char name[50];
  char strasse[30];
  int plz;
  char ort[20];
  float gehalt;
}
main()
{ struct person meier =
  {
    "Franz Meier", "Amselweg 13",
    8900, "Muenchen",
    4200.50
   };
 printf("%s verdient %7.2f DM\n", meier.name,
                                  meier.gehalt);
}
```

Listing 2.2.3: So initialisiert man eine Struktur

Natürlich muß vor main() die Typdefinition wie in Listing 2.2.1 stehen, was ich betonen muß. Nachher werden Sie nämlich die Definitionen der Strukturen nur noch als Include-Files von Diskette laden. Und wenn da beispielsweise der Typ *NewWindow* definiert ist und Sie *newwindow* schreiben, meldet der Compiler einen undefinierten Typ. Auch sehr wichtig ist, daß Sie bei dieser Lösung die Struktur komplett und mit den richtigen Typen initialisieren. Würden Sie beispielsweise hier die »8900« vergessen, müßte der Compiler "Muenchen" auf den Platz der int-Komponente *plz* schreiben. Das kann er nicht, also meckert er.

Struktur in der Struktur

Was beim Amiga auch häufig vorkommt, sind Strukturen innerhalb eines *struct*. Das zeigt am besten das Beispiel von Listing 2.2.4.

```
struct punkt
{ int x;
  int y;
};

struct rechteck
{
  struct punkt LinksOben;
  struct punkt RechtsUnten;
};

main()
{
  struct rechteck re;
  re.LinksOben.x = 12;
  re.LinksOben.y = 20;
  printf("%d\n",re.LinksOben.x);
}
```

Listing 2.2.4: »structs im struct« gibt es häufig beim Amiga

Die Aufgabe ist folgende: Es gibt einen Typ *punkt*, der einen Punkt auf dem Bildschirm mit Werten für die X- und die Y-Koordinate beschreiben soll. Dieser Typ *punkt* ist ein *struct* mit Feldern für *x* und *y*. Nun soll die Lage eines Rechtecks auf dem Schirm durch seine linke obere und seine

rechte untere Ecke beschrieben werden. Beide Ecken sind aber auch Punkte. Folglich kann das *struct rechteck* den Typ *punkt* einsetzen.

Ich habe hier auch schon angefangen, Groß- und Kleinschreibung in Bezeichnern zu mischen, weil auch das beim Amiga häufig vorkommt und gerade hier bei Verwechslung der Schreibweisen besonders der Aztec-Compiler massenhaft, aber unpräzise Fehler meldet. Ansonsten gilt hier wieder die Typregel. Weil in einem *struct* jeder Typ erlaubt ist, darf das auch *struct*-Typ sein. Und auch die Referenzregel setzt sich fort: Aus dem Punkt wird »Punkt Punk«.

☞ Im Gegensatz zu Pascal, wo *struct record* heißt, muß ein Typ (hier *punkt*) definiert sein, bevor er (hier in *rechteck*) benutzt wird.

Der Gebrauch von Zeigern

Mit Zeigern kann man in C unheimlich viel anfangen, und einiges davon brauchen wir unbedingt für die Amiga-Programmierung. Also packen wir's an. Die erste Überraschung und Regel lautet:

Die Namen von Arrays sind schon Zeiger.

In Listing 2.2.5 wurde der Array *ar* auf die altbekannte Art angelegt. Weiter gibt es den int-Zeiger *ptr*. Das kennen Sie alles schon. Um *ptr* zu initialisieren, muß man nun aber nicht *ptr=&ar* schreiben, sondern nur noch *ptr=ar*, wie es Listing 2.2.5 zeigt. Das Programm gibt die Zahlen 11, 12 und 13 aus, aber wie? Das schauen wir uns einmal genau an.

```
int ar[] = {11,12,13};
int *ptr;
main()
{
   ptr = ar;
   printf("%d\n", *ptr);
   ptr = ptr + 1;
   printf("%d\n", *ptr);
   ptr = &ar[0];
   ptr = ptr + 2;
   printf("%d\n", *ptr);
}
```

Listing 2.2.5: Zeigerarithmetik braucht man für Arrays

ar ist die Adresse des Arrays *ar*. Da steht auch sein erstes Element, hier die 11. *ptr* zeigt auf diese Adresse, folglich ist **ptr* ihr Inhalt, also die 11. Die wird mit *printf()* ausgedruckt. Nun folgt

```
prt = ptr + 1;
```

Damit wird die Adresse nicht um 1, sondern um 4 Byte erhöht. Das überrascht Sie? Nun, wir hatten schon gelernt, daß Zeiger typgebunden sind. Ein int-Zeiger kann nur auf Integers zeigen. Und hier erkennen Sie auch den Grund für die Typgebundenheit. int-Variablen belegen (ohne besondere Compiler-Option) immer 4 Byte. Als Programmierer müssen Sie das aber nicht unbedingt wissen. Sie sagen sich nur: Wenn ich einen Zeiger um 1 erhöhe, soll er auf die nächste Variable zeigen. Wieviel Bytes der Compiler braucht, um die Variable abzulegen, interessiert mich nicht. Die Korrektur soll der Compiler selber vornehmen.

Wegen der Typgebundenheit wird der Compiler auch nicht *ptr=&ar* ohne Warnung akzeptieren. Er stößt damit zwar auf dieselbe Adresse, das Programm läuft auch richtig, aber *ptr* ist ein int-Zeiger und kein ar-Zeiger. Die einzelnen Elemente des Arrays hingegen sind Integers, weshalb *ptr = &ar[0];* korrekt ist. In *ar[0]*, worauf *ptr* jetzt zeigt, steht die 11, in *ar[2]* die 13. Folglich muß ich nur mit *ptr = ptr + 2;* auf *ar[2]* zeigen, und schon steht in **ptr* die 13.

Zeiger und Strings

Ein String ist ein *char*-Array mit einem 0-Byte am Ende, aber er ist primär ein Array. Wenn aber ein Array-Name ein Zeiger ist, dann kann man das auch so schreiben. Folglich sind gleichwertig

```
char text[]      und      char *text
```

Die Schreibweise *char *text* sieht man sogar ziemlich oft, weshalb ich sie in Listing 2.2.6 einmal anwende.

```
char *text = "Hallo";
char *ptr;

main()
{
  ptr = text;
  while (*ptr != 0)
```

```
    {   printf("%c\n", *ptr);
        ptr++;
    }
}
```

Listing 2.2.6: Die Anwendung von Zeigern und Strings

Das Programm soll »Hallo« ausgeben, jetzt aber die Buchstaben untereinanderschreiben, wofür man sie einzeln packen muß. Ich nehme hier mit der while-Schleife etwas aus der vierten Sitzung vorweg, deshalb registrieren Sie hier nur:

```
while (*ptr != 0)
```

heißt: Solange der Inhalt der Adresse **ptr* nicht 0 ist (am String-Ende steht 0), führe die Befehle in den folgenden geschweiften Klammern aus. Beim ersten Durchlauf wird das »H« gedruckt. Dann wird *ptr* um 1 erhöht und zeigt auf das »a«. Die Schreibweise *ptr++* ist ein empfehlenswertes Kürzel für *ptr=ptr+1*.

Zeiger auf Strukturen

Wir haben schon in der ersten Sitzung gelernt, daß man einer Funktion Argumente übergeben kann, die auch Variable sein dürfen. Sie wissen auch noch, daß diese Argumente dafür auf die Platzhalter (Parameter) der Funktion kopiert werden. Das hat nun leider zur Folge, daß – zumindest solange die Funktion läuft – die Variablen doppelt im Speicher stehen. Bei großen Arrays kann das sehr viel Speicher kosten, und außerdem braucht das Kopieren auch seine Zeit. Deshalb übergibt man nicht den ganzen Array, sondern nur seine Adresse. Das ist auch der Grund, warum in C Array-Namen automatisch die Adressen des Array-Beginns sind.

Das Platz- und Zeitproblem ergibt sich aber auch für Strukturen, die gleichfalls ganz schön groß werden können und es beim Amiga auch sind. Folglich sollte man auch nicht ganze Strukturen, sondern nur ihre Anfangsadressen übergeben, also Zeiger auf die Strukturen. Bei den eingebauten Funktionen des Amiga haben Sie oft gar keine andere Wahl, das müssen Sie also können. Nun denn, Listing 2.2.7 bringt die Lösung.

```
struct person
{
   char name[50];
   char strasse[30];
   int plz;
   char ort[20];
   float gehalt;
};

void display( struct person *ptr  );

main()
{
   struct person meier =
   {
     "Franz Meier", "Amselweg 13",
     8900, "Muenchen",
     4200.50
   };
   display( &meier );
}

void display( struct person *ptr )
{
   printf( "%s verdient %7.2f\n", ptr->name,
                                  ptr->gehalt);
}
```

Listing 2.2.7: So arbeitet man mit struct-Zeigern

Das Programm soll so wie das in Listing 2.2.3 funktionieren, allerdings mit einem Unterschied. Die Ausgabe soll in der Funktion *display()* erfolgen, und dieser Funktion muß ein Zeiger auf die Struktur übergeben werden. Doch zuerst zur Struktur des Programms: Direkt vor *main()* steht die Zeile

```
void display( struct person *ptr  );
```

Das ist der Prototyp der Funktion *display()*, die selbst nach *main()* steht. In der ersten Sitzung haben Sie schon gelernt, daß man mittels der Prototypen sich selbst zur Ordnung zwingen kann, weil es nun dem Compiler

möglich ist, die Argumenttypen auf Richtigkeit zu prüfen. Hier hat das Prototyping aber noch einen Zweck. Die Funktion *main()* übergibt *display()* einen Zeiger auf die Struktur *meier*. Damit können *main()* und *display()* auf dieselbe Variable zugreifen. Wir hatten bisher gelernt, daß eine Funktion nicht auf die lokalen Variablen einer anderen zugreifen kann. Tatsächlich kann man aber genau dies mit Hilfe von Zeigern umgehen. Eine kleine Sperre gibt es aber noch. Diese Funktionen oder ihre Prototypen müssen vor *main()* stehen. Wenn Sie im Listing 2.2.7 den Prototyp weglassen, müssen Sie die Funktion *display()* vor *main()* setzen.

Nun zu den Zeigern. Im Funktionskopf von *display()* steht

```
void display( struct person *ptr )
```

Das heißt, die Funktion erwartet einen Zeiger vom Typ *struct person*. Der Zeiger selbst heißt *ptr*, der Stern sagt, daß es ein Zeiger ist. »Zeiger auf« heißt aber auch »Adresse von«, womit der Aufruf *display(&meier);* logisch ist, denn »&« heißt »Adresse von«.

Aus dem Punkt wird -->

Nun waren wir es bisher gewohnt, auf die Strukturkomponenten mit dem Punkt-Operator zuzugreifen, zum Beispiel mit *meier.name*. Da hier der Parameter **ptr* heißt, müßte man jetzt **ptr.name* schreiben. Das geht schief, weil der Punkt-Operator eine höhere Priorität als der Stern hat. C würde also zuerst *ptr.name* entwickeln, aber das wäre die Adresse und nicht der Inhalt. Also muß man C überlisten und *(*ptr).name* schreiben. Das geht tatsächlich – probieren Sie es aus –, doch inzwischen hat man dafür ein Kürzel erfunden, und das heißt *ptr->name*. Da diese Form übersichtlicher und auch üblich ist, müssen Sie sich nur merken: Hat man nur einen Zeiger auf die Struktur, muß man anstatt des Punkt-Operators (.) den Komponenten-Operator (->) nehmen.

Workshop

Checkliste

1. Warum muß man einen C-String immer um ein Element größer deklarieren als der Text lang ist?
2. Wann muß man beim Zugriff auf Strukturelemente ».« und wann »->« schreiben?
3. Warum sollte man einer Funktion möglichst nur einen Zeiger auf eine Struktur übergeben und nicht die Strukturvariable selbst?

Ideen für eigene Übungen

1. Weisen Sie im Listing 2.2.4 allen Komponenten Werte zu, und zeigen Sie das mittels *printf()* in Form einer Postanschrift an (ohne das Gehalt).
2. Erweitern Sie analog zu Punkt 1 die display()-Funktion von Listing 2.2.7.

2.3 Dritte Sitzung: Erzeugen eines leeren Windows

Die Themen dieser Sitzung:

- Libraries
- Windows
- Window-Datenstrukturen
- Window-Flags
- Type-Casting
- #include

Information

Der Umgang mit Datenstrukturen ist das A und O der Amiga-Programmierung. In der vorherigen Sitzung haben Sie die Grundlagen dazu gelernt, hier wenden wir uns den ersten Amiga-Strukturen zu. Dabei gibt es ein Problem: Damit die Programme laufen, müssen wir mit *if* und *for* etwas aus der nächsten Sitzung vorwegnehmen. Beide Themen werden nur kurz angeschnitten, mehr dazu finden Sie in der vierten Sitzung.

Grundlagen und Eigenschaften von Windows

Windows sind die Basis für alle Aktivitäten auf dem Amiga. Zuerst muß Ihr Programm ein Fenster öffnen, also dieses Gebilde auf dem Schirm erscheinen lassen. Dabei gilt:

a) Wenn ein Fenster geöffnet wird, muß man ihm dabei Eigenschaften mitgeben. Die Größe und die Lage des Fensters gehören sicherlich dazu, aber auch die Gadgets (z.B. die Dinger zum Schließen oder Größe ändern in den Ecken).

b) Hat man einem Fenster ein Gadget (oder mehrere) mitgegeben, muß man auch sagen, ob Intuition dem Programm melden soll, wenn ein Gadget betätigt wurde.

c) Es ist Sache des Programms, auf diese Gadget-Meldungen zu warten und darauf zu reagieren.

Die Punkte a) und b) werden beim Öffnen des Fensters erledigt. Ein Window wird durch einen Aufruf von *OpenWindow;()* geöffnet. Diese Funktion hat nur einen einzigen Parameter, nämlich einen Zeiger auf die NewWindow-Struktur. *NewWindow* ist eine Daten-Struktur, in die man vor dem Aufruf von *OpenWindow()* alle Eigenschaften des Fensters eintragen muß. Die Funktion gibt einen Zeiger auf eine Daten-Struktur mit dem Namen *Window* zurück. In der Window-Struktur steht zuerst eine Kopie der Daten von *NewWindow* aus dem Aufruf, aber auch einiges mehr. Das hört sich komplizierter an, als es ist. Nehmen wir deshalb mit Listing 2.3.1 ein wenig Praxis vorweg.

```
struct NewWindow NeuesWindow =
{
 20,20,300,100,      /* Lage u. Maße              */
 -1,-1,              /* Farben                    */
 CLOSEWINDOW,        /* Melde Klick Close-Gadget  */
 WINDOWCLOSE,        /* Installiere Close-Gadget  */
 NULL,NULL,          /* keine Extras              */
 "Mein Window",      /* Titel-Text                */
 NULL,               /* Kein eigener Screen       */
 NULL,               /* kein SuperBitmap-Window   */
 0,0,0,0             /* keine Größenänderung      */
 WENCHSCREEN,        /* Bildschirmtyp             */
};

main()
{
  Window = OpenWindow(&NeuesWindow);
}
```

Listing 2.3.1: Öffnen eines Fensters (Beispiel ist unvollständig!)

Das Beispiel ist zwar noch etwas unvollständig (läuft so nicht), es verdeutlicht aber schon recht gut das Prinzip von nahezu allen Operationen unter Intuition. Die NewWindow-Struktur zeigt auch eine weitere typische Eigenschaft: Es sind viele Features vorgesehen, die man kaum immer alle braucht. In diesem Fall muß man NULL eintragen, falls es sich um einen Zeiger handelt, oder 0 im Falle von Daten. Für »Farben« steht da zweimal -1, was soviel heißt wie »übernehme« (hier die Farben des Screen). Wichtig ist der Eintrag WBENCHSCREEN. Ein Window braucht immer einen Screen (Bildschirm), der die Auflösung und die Anzahl der mög-

lichen Farben für das Window vorgibt. Ein Screen wird so ähnlich wie ein Window angelegt, doch die Arbeit können wir uns ersparen, wenn wir mit WBENCHSCREEN sagen, daß das Window den Workbench-Screen nehmen soll.

Als guter C-Programmierer ist Ihnen natürlich aufgefallen, daß hier einige Konstanten und Typen nicht definiert sind. Keine Sorge, die beschaffen wir uns später ganz leicht mit einigen #include-Anweisungen.

Libraries

Die im Listing 2.3.1 genannte Funktion *OpenWindow()* ist eine von Hunderten, die im Amiga schon eingebaut sind. Diese Funktionen sind in Gruppen zusammengefaßt, die Libraries (Bibliotheken) heißen. Bevor man so eine Funktion nutzen kann, muß man ihre Library öffnen. Sinn der Übung: einige Libraries stehen nicht im ROM, sondern auf der Disk. Findet der Amiga eine Library nicht im ROM, schaut er im Library-Directory auf der Diskette nach. Das macht den Amiga sehr flexibel, weil man für neue Bertriebssystem-Funktionen nur neue Libraries auf einer Diskette braucht. Hat ein Programm (Task) die Library geladen, können sie andere Tasks auch nutzen. Damit die Library nicht ewig den RAM blockiert, muß jedes Programm zum Schluß seine Library auch wieder schließen. Hat das letzte Programm eine Library geschlossen, gibt das System den Speicher wieder frei.

Da dieser Ablauf allgemein gültig ist, gilt er auch für ROM-Libraries. Und sogar hier müssen Sie nach dem Öffnen mittels der Funktion *OpenLibrary;()* prüfen, ob kein Fehler aufgetreten ist. Daß eine Datei auf der Diskette fehlt, kann man sich ja noch vorstellen, aber kann da plötzlich ein Stück ROM ausfallen? Nun, das wohl nicht. Doch Tatsache ist, daß der Amiga Speicher braucht, wo er sich notiert, daß Ihr Programm diese Library geöffnet hat. Und genau dieser Speicher kann im Extremfall fehlen.

Include-Files

Jedes typische Amiga-Programm beginnt mit Include-Anweisungen, wie beispielsweise den folgenden:

```
#include <exec/types.h>
#include <intuition/intuition.h>
```

#include ist eine Anweisung an den Compiler, die aufgeführte Textdatei – praktisch ein unvollständiges C-Programm – einzulesen. Das wirkt genauso, als ob Sie den Text an dieser Stelle eingetippt hätten. Die Dateien haben zwar den Extender ».h«, und heißen Header-Files, doch *#include* kann durchaus an beliebiger Stelle im Programm stehen, und die Datei muß auch nicht mit ».h« enden.

Wenn der Name wie hier in spitzen Klammern eingeschlossen ist, sucht der Compiler zuerst im Directory, auf das die Include-Umgebungsvariable zeigt. Mit Anweisungen wie »assign INCLUDE: sys:lc/include« hatten wir das System schon im Kapitel 1.5 so eingerichtet. Wenn Sie hingegen den Namen (plus Pfad) in Anführungsstriche setzen, sucht der Compiler zuerst dort. Typisch wendet man diese Form für eigene Include-Files an. In den oben genannten Beispielen werden die Include-Files *types.h* und *intuition.h* eingezogen. Damit werden diverse Typen und Konstanten definiert. Schauen Sie sich ruhig einmal mit einem Editor diese Dateien an.

Praxis

Mit den Include-Direktiven sind wir auch schon mitten in der Praxis des Listings 2.3.2. Mit der dann folgenden Zeile

```
struct IntuitionBase *IntuitionBase;
```

wird eine Variable vom Typ »Zeiger auf die Struktur IntuitionBase« deklariert. Der Typ *IntuitionBase* steht im Include-File. Beachten Sie, daß es durchaus zulässig und auch üblich ist, bei selbstdefinierten Typen für den Typ und die Variable den gleichen Namen zu verwenden. Die Regelung ist sinnvoll, da man ja bei der Namensvergabe nicht wissen kann, welche Typen in den Include-Files stehen. Bei den C-Standardtypen geht das nicht. Also *int int* dürfen Sie nie schreiben, wohl aber das:

```
typedef int ganz;
main()
{ ganz ganz;
  ganz = 13;
  printf("%d",ganz);
}
```

Hier habe ich eine Variable mit dem Namen *ganz* vom Typ *ganz* geschaffen. Mit *typedef* der Syntax *Synonym* kann man in C Typen neue Namen geben, was meistens zum Ausgleich von Compiler-Unterschieden gebraucht wird.

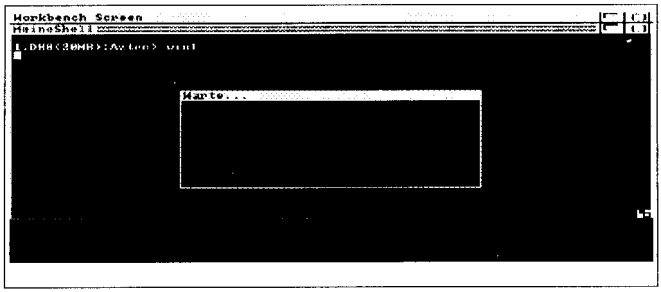


Bild 2.3.1: Das ist unser Ziel

Doch zurück zum Listing 2.3.2. Auch der Struktur-Typ *struct NewWindow* steht im Inlclude-File *intuition.h*. Wir müssen jetzt die Struktur-Komponenten mit Werten laden, was derart geschieht, daß wir die Variable *MeinWindow* deklarieren und sie dadurch dabei gleichzeitig initialisieren.

Type-Casting muß sein

Dazu muß man natürlich die Typen der einzelnen Strukturkomponenten kennen, drucken Sie einfach *intuition.h* aus. Nur dann finden Sie den Grund für die Zeile

```
(UBYTE *) "Warte...", /* Window-Titel          */
```

Lassen Sie das (UBYTE *) weg, meldet Aztec C eine Pointer-Pointer-Konvertierungs-Warnung, Lattice beanstandet das nicht. Der Grund ist, daß in *intuition.h* die Komponente als »UBYTE *Title« deklariert ist, d.h., **Title* ist ein UBYTE-Zeiger. UBYTE (aus dem Include-File *exectypes.h*) ist ein mit *typedef* definiertes Kürzel für den Typ *Byte* ohne Vorzeichen. Das ist zwar im Prinzip der Typ *char* auch, nur erzeugt ein String-Literal wie hier das "Warte...", einen char-Zeiger und keinen UBYTE-Zeiger.

```
/* win1.c
   Programm öffnet ein Window, läßt es für einige
   Sekunden auf dem Schirm und endet dann.
*/
#include <exec/types.h>
#include <intuition/intuition.h>
struct IntuitionBase *IntuitionBase;
/* Struktur für ein neues Window initialisieren: */
```

Listing 2.3.2: (Fortsetzung nächste Seiten)

```
struct NewWindow MeinWindow =
{
   170,80,                 /* linke obere Ecke      */
   300,100,                /* Breite u. Höhe        */
   -1,-1,                  /* Farbe der Pens        */
   0L, 0L,                 /* Keine Flags           */
   NULL,                   /* keine User-Gadgets    */
   NULL,                   /* keine User-CheckMark  */
   (UBYTE *)"Warte...",    /* Window-Titel          */
   NULL,                   /* Kein eigener Screen   */
   NULL,                   /* keine SuperBitmap     */
   100,                    /* Mindestbreite         */
   30,                     /* Mindesthöhe           */
   640,                    /* Maximalbreite         */
   256,                    /* Maximalhöhe           */
   WBENCHSCREEN            /* Bildschirmtyp         */
};

void _main() /* Beachte Unterstrich vor main */
{
 struct Window *Window;
 long i;

 /* Öffne Intuition-Library: */
 IntuitionBase = (struct IntuitionBase *)
               OpenLibrary("intuition.library",0L);

 /* Abbruch, wenn Fehler: */
 if (IntuitionBase == NULL)  exit(FALSE);

 /* Öffne Window: */
 Window = (struct Window *) OpenWindow(&MeinWindow);
 if (Window == NULL) {           /* Wenn Fehler:  */
   CloseLibrary(IntuitionBase); /* Lib schließen */
   exit(FALSE);                 /* und Abbruch   */
  }
 for(i=1000000; i; i--);      /* Warteschleife */
```

Listing 2.3.2: (Fortsetzung nächste Seite)

```
 CloseWindow(Window);          /* Window schließen */
 CloseLibrary(IntuitionBase); /*Library schließen */
 exit(TRUE);                   /*      Ende        */
}
```

Listing 2.3.2: Die einfachste Art, ein Window zu öffnen

Generell geht es hier darum, einer Variablen vom Typ *x* ein Datum vom Typ *y* zuzuweisen. Da der C-Compiler nicht wissen kann, ob das ein Versehen mit teilweise schlimmen Folgen oder Absicht ist, müssen wir ihm das sagen, sprich, den einen Typ ausdrücklich in den anderen umwandeln. Da dieses »Type-Casting« sehr häufig gebraucht wird, müssen wir das üben. Listing 2.3.3 hat diesen Zweck.

```
main()
{
   int i = 90;
   char c = 'c';
   char text[] = "ABC" ;

   printf("%c \n", (char) i);
   printf("%d \n", (int) c);
   text[2] = (char) 65;
   printf("%s \n", text);
}
```

Listing 2.3.3: Type-Casting ist wichtig in Amiga C

Das Programm gibt der Reihe nach in drei Zeilen Z, 99 und ABA aus. Im Rechner werden auch Buchstaben und andere Zeichen als Zahlen gespeichert. In einer Norm namens ASCII ist festgelegt, welche Zahl welchem Zeichen entspricht. Demnach sind Zahlen und Zeichen austauschbar, was hier angewandt wird.

»printf;("%c \n", (char) i)« soll laut dem Formatstring ein Zeichen drucken, wir geben aber die int-Variable *i* aus. Für die Umwandlung in ein Zeichen wird vor das i *(char)* gesetzt. Umgekehrt soll »printf("%d \n", (int) c);« laut Formatstring einen int-Wert ausgeben, wir bieten ihm aber ein Zeichen an. Das wird durch den Vorsatz (int) umgewandelt. Die näch-

ste Zeile setzt in den String ABC für das C die Zahl 65 ein, was der ASCII-Code von A ist. Sie sehen, daß man den gewünschten Typ in Klammern vor den fremden setzen muß.

Diese Regel gilt generell auch für Zeigertypen, nur daß man hier nach dem Namen noch einen Stern setzen muß. Schauen wir uns Listing 2.3.4 an.

```
typedef unsigned char UBYTE;

main()
{
   UBYTE byte_array[4] = {65,66,67,0};
   char *char_zeiger;

   char_zeiger = (char *) byte_array;
   printf("%s\n", char_zeiger);

   strcpy(byte_array, (UBYTE *) "xyz");
   printf("%s\n", (char *) byte_array);
}
```

Listing 2.3.4: Typwandlung bei Zeigern

Das hier vorliegende Problem ist an sich trivial und wird von vielen Compilern auch nicht mehr angemeckert. Der Typ *char* ist im Standard-C als *signed char* (signed = Vorzeichen) mit dem Wertbereich −128 bis +127 definiert. Der Amiga hat jedoch viele Sonderzeichen mit den ASCII-Codes von 128 bis 255. Deshalb hat man *unsigned char* mit dem Wertbereich 0 bis 255 als den Typ UBYTE definiert. Die erste Zeile von Listing 2.3.3 steht so im Include-File */exec/types.h*.

In Listing 2.3.4 wird *byte_array* (vom Typ UBYTE) mit {65,66,67,0} initialisiert. Das sind die ASCII-Codes für ABC gefolgt von einem Null-Byte, also ist das ein String in der rechnerinternen Darstellung. Bei einem Array ist der Name gleichzeitig die Adresse, also kann man ihm eine Zeigervariable zuweisen. Das geschieht mit »char_zeiger = (char *) byte_array;«. Dabei muß der *byte_array* vom Typ UBYTE in den Typ *char* gewandelt werden. Zusätzlich müssen wir aber noch sagen, daß es sich um einen Zeiger handelt, und das tut der Stern in *(char *)*.

Nun zur Umkehr: Mit der Zeile

```
strcpy(byte_array, (UBYTE *) "xyz");
```

wird der String *xyz* in den Bytearray kopiert. Weil *xyz* ein String und damit ein char-Zeiger ist, muß er in einen UBYTE-Zeiger gewandelt werden. Die Regel (Typ *) gilt für alle Zeiger, auch solche, die auf Strukturen zeigen. Das erklärt diese Zeilen:

```
struct Window *Window;
Window = (struct Window *) OpenWindow(&MeinWindow);
```

Die Variable **Window* ist ein Zeiger vom Typ *struct Window*. Die Funktion *OpenWindow* hingegen bringt einen allgemeinen Zeiger. Folglich muß man diesen mittels *(struct Window *)* in einen Window-Zeiger umwandeln.

Wir müssen nur immer öffnen und schließen

Nachdem wir nun das schmückende Beiwerk der C-Funktionen entschlüsselt haben, können wir uns wieder der Aufgabe, ein Fenster zu öffnen, und damit Listing 2.3.2 zuwenden. Wir öffnen die *intuition.library* und merken uns deren Adresse in *IntuitionBase*. Das erste Argument ist der Name der Library, das zweite die Versions-Nummer. Trägt man dafür *0L* ein, heißt das »Version ist mir egal«. Nun folgt die Zeile

```
if (IntuitionBase == NULL)  exit(FALSE);
```

Im Klartext: Wenn der Zeiger *IntuitionBase* den Wert NULL hat (keine gültige Adresse), breche das Programm ab. Mehr zum if-Befehl finden Sie in der vierten Sitzung. Die Library mußten wir öffnen, weil wir die darin befindliche Funktion *OpenWindow()* brauchen.

OpenWindow() hat nur ein Argument, nämlich die Adresse einer NewWindow-Struktur. Diese haben wir vorher mit den vielen Zeilen als *MeinWindow* initialisiert. Ging beim Öffnen etwas schief, was mit »if (Window == NULL)« abgefragt wird, müssen wir alles, was offen ist, schließen. Das ist hier aber nur die *intuition.library*. Folglich können wir das mit *CloseLibrary(IntuitionBase)* erledigen. Sie sehen, die Funktion will einen Zeiger auf die Library sehen, also gut, daß wir uns den in *IntuitionBase* gemerkt hatten.

Läßt sich das Window öffnen, ist es auch auf dem Schirm. Damit Sie sich das Kunstwerk ein paar Sekunden lang ansehen können, läuft jetzt die for-Schleife los. Sie zählt nur von 1 000 000 bis 0, was Zeit braucht, mehr

dazu in der vierten Sitzung. Ist das Programm bis hierher gelaufen, sind das Window und die Library offen, also müssen wir beide schließen. Nun können wir mit *exit()* das Programm verlassen.

☞ Beachten Sie, daß ich nicht *main*, sondern *_main* (mit Unterstrich) geschrieben habe. Der Linker wird deshalb einen kürzeren Startup-Code einbinden, der nicht mehr die Standard-I/O-Kanäle öffnet. Damit sind aber auch Funktionen wie *printf()* nicht mehr verwendbar. Diese werden allerdings auch nicht gebraucht.

Wir rationalisieren die Arbeit

Sie haben es schon erkannt: Ein echtes Amiga-Programm ohne Windows gibt es nicht. Also müssen wir immer die Libraries öffnen, dann Intuition, dann die Windows, und zum Schluß müssen wir alles wieder schließen. Und das sollen wir immer tippen? Natürlich nicht. Wir schaffen uns ein paar Funktionen für diese Arbeiten, packen diese in ein Include-File und laden dieses dann nur noch, wenn wir ein neues Programm anfangen. Listing 2.3.5 arbeitet schon auf dieser Basis.

```
/* win2.c */

#include <exec/types.h>
#include <intuition/intuition.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;

#include "stdwindow.h" /* Eigener File! */

void _main(void)
{
   open_libs(); /* Libraries öffnen */

   Window = (struct Window *) open_window(
            20,20,500,100," Der Fenstertitel ",
            WINDOWCLOSE | WINDOWSIZING | WINDOWDRAG,
            CLOSEWINDOW, NULL);
```

Listing 2.3.5: (Fortsetzung nächste Seite)

```
    if (Window == NULL) exit(FALSE);

    /* Warte auf Mausklick in Close-Gadget */
    Wait(1L << Window->UserPort->mp_SigBit);

    close_all(); /* Alles schließen */
}
```

Listing 2.3.5: Das rationalisierte Window-Programm

Der Anfang ist noch klar. Es werden zwei Inlude-Files eingelesen und drei Variablen deklariert. Stören Sie sich nicht daran, daß wir **.GfxBase* noch nicht nutzen, das Include-File braucht sie schon jetzt, wir wenden es später an. Nun folgt

```
#include "stdwindow.h" /* Eigener File! */
```

Diese Datei finden Sie in Listing 2.3.6. Sie müssen zuerst dieses Listing eintippen, es unter dem Namen *stdwindow.h* speichern und erst dann Listing 2.3.5 eingeben, es kompilieren und testen. Sie sollten *stdwindow.h* im selben Directory wie Ihre Programme ablegen. Dieses Header-File stellt drei Funktionen zur Verfügung, nämlich *open_libs()* zum Öffnen der erforderlichen Libraries, *open_window()* zum Öffnen eines Window, und *close_all()* zum Schließen des Windows und der Libraries. Damit reduziert sich unser Programm – vom Vorspann abgesehen – auf die Zeilen

```
open_libs();
open_window(...);
Wait(1L << Window->UserPort->mp_SigBit);
close_all();
```

Die vorletzte Zeile mit der *Wait;()*-Funktion bringt Sie sicherlich ins Grübeln, doch glauben Sie mir vorerst, daß dadurch das Programm solange wartet, bis das Close-Gadget des Window angeklickt wird. In der fünften Sitzung gehen wir der Sache auf den Grund. Zuerst müssen wir uns *open_window()* laut Listing 2.3.6 ansehen.

```
/***************** stdwindow.h ********************/
void open_libs(void)
{
    IntuitionBase = (struct IntuitionBase *)
```

Listing 2.3.6: (Fortsetzung nöchste Seiten)

```
                OpenLibrary("intuition.library",0L);
   if (IntuitionBase == NULL)  Exit(FALSE);

   GfxBase = (struct GfxBase *)
             OpenLibrary("graphics.library",0L);
   if (GfxBase == NULL)
   {
    CloseLibrary(IntuitionBase);  /* Int.-Lib zu */
    Exit(FALSE);                  /* Abbruch     */
   }
}
struct Window *open_window (short x, short y,
                            short w, short h,
                            char *name,
                            ULONG flags,
                            ULONG i_flags,
                            struct Gadget *gadget )
{
   struct NewWindow nw;

 nw.LeftEdge = x;     /* linke Kante des Fensters */
 nw.TopEdge = y;      /* obere Kante des Fensters */
 nw.Width = w;        /* Breite des Fensters      */
 nw.Height = h;       /* Höhe des Fensters        */
 nw.DetailPen = -1;
 nw.BlockPen  = -1;
 nw.Title = (UBYTE *) name;    /* Fenster-Titel  */
 nw.Flags = flags;             /* Welche Gadgets */
 nw.IDCMPFlags = i_flags;      /* Welche IDCMP's */
 nw.Screen = NULL;
 nw.Type = WBENCHSCREEN;
 nw.FirstGadget = gadget;
 nw.CheckMark = NULL;
 nw.BitMap =  0;
 nw.MinWidth =  -1; nw.MinHeight = -1;
 nw.MaxWidth =  -1; nw.MaxHeight = -1;

 return( (struct Window *) OpenWindow(&nw) );
```

Listing 2.3.6: (Fortsetzung nöchste Seite)

```
}

void close_all(void)
{
   CloseWindow(Window);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);
   Exit(TRUE);
}
```

Listing 2.3.6: Das Include-File stdwindow.h

Bisher haben wir die NewWindow-Variable gleich bei der Deklaration initialisiert. Jetzt legen wir mit

```
struct NewWindow nw;
```

eine Variable mit dem Namen *nw* vom Typ *NewWindow* an. Da es sich dabei um ein *struct* handelt, sind Zuweisungen an die Struktur-Komponenten in dieser Form möglich:

```
nw.LeftEdge = 20;
nw.TopEdge  = 20;
/* u.s.w. */
```

Die Methode erscheint Ihnen im Vergleich zur bisherigen zu umständlich? Sie haben ja recht, aber das Verfahren hat Vorteile. Die initialisierten Strukturvariablen müssen immer global sein, sie sind also auch statisch. Statische Variable belegen immer ihren Speicherplatz und können nicht gelöscht werden.

Nun müssen Sie wissen, daß die NewWindow-Struktur nur zum Öffnen des Windows benötigt wird, danach ist sie überflüssig. Alle weiteren Zugriffe auf das Window laufen über die Window-Struktur, ein *struct*, das Intuition nach dem Öffnen automatisch anlegt. Die von *OpenWindow()* gelieferte Adresse zeigt darauf. In der Window-Struktur befinden sich alle Daten aus Ihrer NewWindow-Struktur und noch ein paar mehr.

Jetzt stellen Sie sich vor, Sie brauchen drei Windows, dann würden Sie nach der statischen Methode auch dreimal den Platz für die NewWindow-Strukturen unnötig belegen. Folgerung: Wir legen die ganze »NewWindow-Geschichte« in eine Funktion. Dort deklarierte Variablen sind bekanntlich dynamisch (automatische Variable) und existieren somit

nur so lange (auf dem Stapel), wie die Funktion aktiv ist. Bei der Gelegenheit übergeben wir dann die Eigenschaften des Windows, die wir ändern möchten, als Argumente. Die Funktion weist diese Werte der NewWindow-Struktur zu – die übrigen als Konstanten auch – und öffnet dann das Fenster. Den Zeiger auf die (große) Window-Struktur gibt sie zurück. Beide Aktionen (Window öffnen und Zeiger darauf »returnen«) werden mit der Zeile »return((struct window *) OpenWindow(&nw));« erledigt. Betrachten wir den Aufruf aus Listing 2.3.5, so steht da:

```
Window = (struct Window *) open_window(
        20,20,500,100," Der Fenstertitel ",
        WINDOWCLOSE | WINDOWSIZING | WINDOWDRAG,
        CLOSEWINDOW, NULL);
```

Die ersten vier Zahlen geben die Lage des Fensters sowie seine Größe an. Nach dem Titel des Window folgt als ein Argument der Ausdruck WINDOWCLOSE | WINDOWSIZING | WINDOWDRAG. Es handelt sich hierbei um die Flag-Bits (in *intuition.h* definierte Konstanten), die mittels der ODER-Funktion (|) zusammengefaßt werden. Damit werden die Gadgets für das Schließen, die Größenänderung und das Verschieben eingeschaltet. Probieren Sie diese Gadgets aus, sie funktionieren automatisch. Es gibt noch das Gadget WINDOWDEPTH für das »in den Hintergrund bringen« und weitere, die wir später kennenlernen werden.

Workshop

Checkliste

1. Wann braucht ein Programm einen eigenen Screen?
2. Kann man die NewWindow-Struktur nach dem Öffnen eines Windows vergessen oder gar löschen?
3. Warum sollte man eine Library zum Schluß immer schließen?

Ideen für eigene Übungen

Ändern Sie das Listing 2.3.5 so, daß

- das Window einen neuen Titel trägt,
- es eine andere Lage und Größe hat und
- es alle Gadgets hat (auch WINDOWDEPTH).

Kompilieren, linken und testen Sie das Programm.

2.4 Vierte Sitzung: Den Programmfluß kontrollieren

Die Themen dieser Sitzung:

- Entscheidungen
- Operatoren
- Verzweigungen
- Schleifen
- Sprünge

Information

Sie haben es schon in der vorherigen Sitzung gemerkt: Wir müssen prüfen, ob eine bestimmte Bedingung gegeben ist, und danach das eine oder das andere tun. Das Programm muß in die eine oder in die andere Richtung verzweigen, der Programmfluß wird geändert. Neben diesen einfachen Verzweigungen gibt es auch Mehrfachverzweigungen, die ebenfalls sehr wichtig sind. Zum Beispiel muß ein Programm ganz verschiedene Dinge tun, je nachdem, welcher Menüpunkt ausgewählt wurde. Was gewählt wurde, muß das Programm natürlich erfragen. Und wenn nichts gewählt wurde? Dann muß es wieder fragen und wieder fragen, so lange, bis der Anwender etwas tut. In diesem Fall läuft das Programm in einer Schleife. Aus allem zusammen ergeben sich diese Themen:

- Operatoren in C
- Die verschiedenen Arten der Verzweigung
- Endlose, abweisende und nicht abweisende Schleifen
- Das Bilden von Blöcken

Arithmetische Operatoren einsetzen

Bis jetzt habe ich immer brav wie in Pascal oder Basic *i=i+2* geschrieben, wenn *i* um 2 erhöht werden sollte. Aber das tut man nicht in C, sondern schreibt dafür *i+=2*. Das gilt nicht nur für 2, sondern allgemein und auch für andere Operatoren, wie die Tabelle 2.4.1 zeigt.

Ausdruck	Kurzform	Operation
x = x * y	x * = y	Multiplikation
x = x / y	x / = y	Division
x = x % y	x % = y	Modulo
x = x + y	x + = y	Addition
x = x − y	x − = y	Subtraktion
x = x << y	x << = y	Links schieben
x = y >> y	x >> = y	Rechts schieben
x = x & y	x & = y	Bitweises AND
x = x ^ y	x ^ = y	Bitweises XOR
x = x | y	x | = y	Bitweises OR

Tabelle 2.4.1: Operatoren und ihre Kurzformen

Früher galt, daß man generell die Kurzform verwenden sollte, weil der Compiler dann weiß, daß er die Variable nicht zweimal zwischenspeichern muß. Moderne Compiler erkennen das aber und wandeln selbständig Ausdrücke wie *x=x+17* in *x+=17* um.

Inkrement- und Dekrement-Operatoren

Für den Sonderfall *x+=1* kann man auch *x++* schreiben, womit wir bei den berühmten C-Operatoren ++ und −− angelangt wären. Es gilt:

Ausdruck	Ist gleichwertig mit
x = x + 1	x++ oder ++x (Inkrement)
x = x − 1	x−− oder −−x (Dekrement)
x + = 1	x++ oder ++x
x − = 1	x−− oder −−x

Tabelle 2.4.2: Inkrement- und Dekrement-Operatoren

Die Aussage, daß *x++* und *++x* gleichwertig sind (bzw. *x−−* und *−−x*), ist allerdings nur im Fall der Zuweisungen gültig. Man kann aber auch praktisch an jeder Stelle im Programm Variable mit den Inkrement- bzw. Dekrement-Operatoren schmücken, und dann muß man beide Formen sehr wohl unterscheiden. Das folgende Programm gibt 3 und 5 aus.

```
main() {
    int i=3;
    printf("%d\n", i++);
    printf("%d\n", ++i);
}
```

Die Variable *i* hat den Anfangswert 3. Die erste printf()-Zeile gibt *i* aus und erhöht es dann um 1, *i* ist also jetzt gleich 4. Die zweite printf()-Zeile erhöht erst *i* (von 4 auf 5) und druckt es dann. Das nennt man Post-Inkrement (i++) oder Prä-Inkrement (++i). Für das Dekrement gilt das sinngemäß.

Praxis

Es folgen jetzt viele kleine Programme, welche die Anwendung der verschiedenen Befehle zeigen. Sie sollten alle zum Laufen bringen, sie variieren und wieder testen. Nur Übung macht den Meister.

Verzweigungen mit if und logische Operatoren

Listing 2.4.1 bringt zuerst mit der Funktion *input()* etwas Wiederholung. Die Aufgabe: So ähnlich, wie man in Basic »INPUT "Gebe Zahl ein",i« schreiben darf, wollen wir in C »input("Gebe Zahl ein", &i)« anwenden. Sie sehen, wir übergeben mit *&i* die Adresse von *i*. Das hat zur Folge, daß die Funktion auf diese Adresse schreibt, wir also so einen Wert für *i* zurückbekommen. Beachten Sie auch, daß jetzt in *scanf()* nicht *&zahl*, sondern nur *zahl* steht, denn im Kopf haben wir *zahl* bereits als Zeiger deklariert.

```
void input(char *msg, int *zahl)
     { printf("%s ",msg);
       scanf("%d", zahl);
     }

main()
{
  int i;

  input("Gebe 1 oder 2 ein", &i);
  if (i == 1)
    puts("Das war die 1");
```

```
  if (i == 2)
    puts("Das war die 2");
  if ( i != 1 && i != 2 )
    {
     puts("Tut mir leid");
     printf("%d ist weder 1 noch 2\n", i);
    }
}
```

Listing 2.4.1: Die Anwendung von if

Nun zur Aufgabe: Der Anwender soll 1 oder 2 eingeben, und das soll bestätigt werden. Bei jeder anderen Zahl soll die Ausgabe lauten »xxx ist weder 1 noch 2«. Nun, die Zahl steht nach dem Aufruf unserer input()-Funktion in i, und jetzt kommt die Abfrage

```
if (i == 1)
```

auf deutsch: »wenn i gleich 1 ist«. Beachten Sie, daß der logische Operator »gleich« mit »==« (zwei Gleichheitszeichen) geschrieben werden muß! Wenn diese Bedingung wahr ist (zutrifft), wird der folgende Befehl oder Befehlsblock (siehe unten) ausgeführt. Hier wird mit der Funktion *puts()* (put string) ein Text ausgegeben. Wenn Sie Text unformatiert ausgeben wollen, ist diese Form schneller als *printf()*.

☞ Beachten Sie: Nach *if* darf kein Semikolon stehen. Es wirkt in diesem Fall als leerer Befehl. Das heißt, wenn die Bedingung erfüllt ist, wird dieser leere Befehl und nicht die Folgezeile ausgeführt.

Nun zur Zeile mit dem tollen Ausdruck

```
if ( i != 1 && i != 2 )
```

Im Klartext: »wenn i ungleich 1 und i ungleich 2«. Sie sehen, daß != der Ungleichoperator ist, und && für das logische UND steht.

☞ Beachten Sie, daß C im Gegensatz zu Basic oder Pascal zwischen logischen Operatoren und Bit-Operatoren unterscheidet. Das logische UND heißt *&&*, das bitweise nur *&*.

Nach der Abfrage sollen zwei Zeilen ausgeführt werden, weshalb hier mittels der geschweiften Klammern ein Block gebildet wird. Ein Block darf beliebig viele Zeilen und auch weitere Abfragen enthalten. Natürlich darf er auch dem *if* folgen.

Andernfalls oder else einsetzen

Listing 2.4.2 ist eine kleine Abwandlung eines Teils von Listing 2.4.1. Den Rest übernehmen Sie bitte. In der ersten Abfrage prüfen wir auf »i gleich 1 ODER i gleich 2«. Zwei senkrechte Striche heißen logisch ODER, nur einer meint auch hier nur bitweises »Odern«.

```
  input("Gebe 1 oder 2 ein", &i);
  if (i == 1 || i == 2)
    puts("Das war 1 oder 2");
  else
    {
     puts("Tut mir leid");
     printf("%d ist weder 1 noch 2\n", i);
    }
}
```

Listing 2.4.2: Die Anwendung von else (Auszug)

Ist die if-Bedingung nicht erfüllt, kommt *else* (andernfalls) zum Tragen. Würden Sie das *else* weglassen, hätten Sie einen typischen Logik-Fehler, wonach das Programm einmal korrekt und einmal falsch arbeitet. Gäben Sie dann 55 ein, käme die richtige Meldung »Tut mir leid / 55 ist weder 1 noch 2«, doch würden Sie 1 eingeben, lautete die Ausgabe:

```
Das war 1 oder 2
Tut mir leid
1 ist weder 1 noch 2
```

Mehrfachverzweigungen mit switch erledigen

Im Listing 2.4.1 hatten wir mit der Folge

```
if (i == 1)
  puts("Das war die 1");
if (i == 2)
  puts("Das war die 2");
```

auf nur zwei Werte geprüft, bei 10 verschiedenen Zahlen dürfte diese Methode langsam lästig werden. Deshalb gibt es dafür eine Kurzform namens *switch*. Listing 2.4.3 zeigt die Anwendung.

```
void input(char *msg, int *i)
{ printf("%s ",msg);
  scanf("%d", i);
}

main()
{
  int i;
  input("Gebe 1, 2, 3 oder 7 ein", &i);
  switch(i)
  {
    case 1:
       puts("Das war die 1");
       break;
    case 2:
       puts("Das war die 2");
       break;
    case 3:
       puts("Das war die 3");
       break;
    case 7:
       puts("Das war die 7");
       puts("so ein Glück");
       break;
    default:
       puts("War nicht 1,2,3 oder 7");
  }
}
```

Listing 2.4.3: switch ist die Kurzform für viele if

Per Prinzip schreibt man zuerst diese Zeilen:

```
switch(i)
{

}
```

Hier ist *i* die Variable, die abgefragt werden soll. Da auf *switch* immer ein Block folgt, schreibe ich die geschweiften Klammern schon einmal hin, um dann erst die Zeilen dazwischen einzusetzen. Diese Technik wende ich

immer an – auch bei *main() {...}* –, weil ich so nicht mitzuzählen brauche, wieviel Klammern ich noch schließen muß. Im switch-Block steht nun für jeden Fall ein case-Befehl, dem alle zugehörigen Zeilen folgen. Nach der letzten Zeile eines *case* folgt *break*. Die Klammern sind hier für die Blockbildung nicht erforderlich.

Aussteigen mit break

Mit dem Break-Befehl wird der aktuelle switch-Block verlassen. Das funktioniert so:

```
switch(i)
{
   ....
   ....
  break;
   ....
   ....
}
/* Hier geht es nach break weiter */
```

☞ Mit *break* können Sie nur aus switch-Blöcken und Schleifen aussteigen. Verlassen wird immer nur der aktuelle Block und nicht etwa bei Schachtelungen gleich alles.

Schleifen bilden mit while, do und for

In einer Schleife wird ein bestimmter Programmteil – Schleifenkörper genannt – wiederholt. Eine Form sähe so aus:

```
Beginn:
  tue dieses
  tue jenes
goto Beginn
Ende:
```

Die beiden »tue«-Anweisungen sind der Schleifenkörper. Wie leicht zu erkennen, läuft diese Schleife endlos, die Marke »Ende:« wird nie erreicht. Im Normalfall prüft man deshalb in der Schleife auf eine bestimmte Bedingung, zum Beispiel darauf, ob die Taste [x] gedrückt wird, und springt dann zu »Ende:«, also aus der Schleife heraus. Wir können dafür ein Beispiel in C so formulieren:

```
main()
{
  Beginn:
    puts("Ende mit Taste x");
    if (getch() == 'x')
      goto Ende;
  goto Beginn;
  Ende:
  ;
}
```

Beachten Sie den kleinen Trick mit der Funktion *getch()* (get character, warte auf Tastendruck). Anstatt

```
c = getch();
if (c == 'x')
```

kann man auch gleich mit dem Funktionswert arbeiten. Sie sehen aber auch schon, daß diese Lösung mit *goto* sehr unübersichtlich ist, weshalb C dafür Besseres bietet. Das schauen wir uns mit Listing 2.4.4 an.

```
main()
{ int test;

  test = 0;
  while (test < 10)
  {
    printf("%d\n",test);
    test += 2;
  }

  test = 0;
  do
 {
    printf("%d\n",test);
    test += 2;
  } while(test < 10);
}
```

Listing 2.4.4: Die Anwendung von while und do

Beide Schleifen drucken die Zahlen 0 bis 8 in Zweierschritten. Die Zeile

```
while (test < 10)
```

bedeutet: Solange der Wert von *test* kleiner als 10 ist, führe den nachfolgenden Schleifenkörper aus. Dieser kann wie hier ein Block in geschweiften Klammern, aber auch nur eine Zeile sein. Wichtig ist, daß die Bedingung, auf die *while()* prüft, innerhalb des Schleifenkörpers einmal eintritt, sonst läuft die Schleife endlos. Hier wird die Variable *test* bei jedem Durchlauf um 2 erhöht, so daß sie nicht lange < 10 bleiben kann.

Die zweite Schleife mit *do...while* führt hier zum selben Ergebnis, dennoch gibt es einen wesentlichen Unterschied. Ändern Sie nämlich in beiden Fällen den Ausgangszustand *test=0* in zum Beispiel *test=20*, so wird die while-Schleife nicht, die do-Schleife hingegen einmal ausgeführt (sie gibt die 20 aus). Der Grund ist klar. *while* prüft vor dem Schleifenkörper, *do* hingegen erst danach. Die erste Form nennt man abweisende, die zweite nicht abweisende Schleife.

```
main()
{ int test = 0;
  while (1)
  {
    printf("%d\n",test);
    test += 2;
    if (test > 10)
      break;
  }
  puts("Schleife ist abgelaufen");
}
```

Listing 2.4.5: while endlos mit break-Ausgang

Listing 2.4.5 zeigt eine andere Lösung derselben Aufgabe. Bei der while-Bedingung kommt es letztlich immer darauf an, ob der Ausdruck zu logisch wahr (TRUE) oder falsch (FALSE) entwickelt werden kann. Praktisch ist FALSE als Null definiert und TRUE als jeder Wert ungleich Null. Folglich ist in *while(1)* der Ausdruck immer wahr, die Schleife liefe endlos. Hier erfolgt jedoch die Abfrage innerhalb der Schleife mit *if* und gegebenenfalls der Abbruch mit dem oben geschilderten *break*. Diese Form der while-Schleife findet man recht häufig, weil es oft nötig ist, auf verschie-

dene Abbruchbedingungen zu prüfen. Probieren Sie das Programm auch einmal mit *while(0)* aus.

Wie schon gesagt, kann in der while-Bedingung jeder beliebige Ausdruck eingesetzt werden, wie Listing 2.4.6 zeigt.

```
main()
{
  char s[80];
  while( strcmpi(s,"ende") )
   {
     puts("Gebe ende ein");
     gets(s);
   }
}
```

Listing 2.4.6: while-Ausdrücke sind viele erlaubt

Die Stringvergleichsfunktion *strcmpi()* ergibt nur dann Null (FALSE), wenn beide Strings gleich sind. Folglich läuft die while-Schleife so lange, bis »ende« eingetippt wird.

Wir zählen mit for

Im Listing 2.4.4 hatten wir eine while-Schleife genutzt, um numerische Variable hochzuzählen. Da dieser Fall ziemlich oft vorkommt, bietet C dafür eine Sonderlösung namens *for*. Das können andere Sprachen auch, aber keine hat die »for-Power« von C. Fangen wir einmal mit Listing 2.4.7 ganz harmlos an.

```
main()
{
  int test;
  for(test=0; test<10; test += 2)
   printf("%d\n",test);
}
```

Listing 2.4.7: Unsere erste for-Schleife

Das Beispiel hat dasselbe Ergebnis wie die while-Lösung in Listing 2.4.4. In einer for-Schleife sind drei Ausdrücke möglich, nämlich – in dieser Reihefolge – je einer für Initialisierung (den Startwert), die Testbedingung

und die Modifikation. Die Variable *test* bekommt den Startwert 0. Solange *test < 10* ist, soll die Schleife laufen. In jedem Durchgang soll *test* – auch die Laufvariable genannt – um 2 erhöht werden. In diesem Beispiel besteht der Schleifenkörper nur aus einer Zeile, er darf aber auch ein Block (in geschweiften Klammern) sein.

Wir werden trickreich

Die drei Ausdrücke der for-Schleife erlauben viele Varianten und Tricks. Der erste Trick an der Geschichte ist, daß die Ausdrücke auch leer sein dürfen. Das gleiche Ergebnis wie Listing 2.4.7 bringt diese Lösung:

```
int test=2;
for( ; test<10; test += 2)
```

Hier fehlt der Startwert, und das klappt sogar, weil er vorher schon mit *int test=2* initialisiert wurde. Beachten Sie aber, daß das Semikolon nicht fehlen darf. Auch die Modifikation kann man weglassen, wenn man sie in den Schleifenkörper verlegt. Das sähe dann so aus:

```
int test=2;
for( ; test<10; )
 {
    printf("%d\n",test);
    test += 2;
 }
```

Sie können alles weglassen und *for(;;)* schreiben. In diesem Fall läuft die Schleife endlos. Häufig wird das genutzt und dafür sogar – um das deutlich zu machen – ein Makro namens FOREVER (für immer) definiert.

```
#define FOREVER for(;;)
main()
{
  FOREVER
  {
     puts("Ende mit x");
     if (getch()=='x')
       break;
  }
}
```

Listing 2.4.8: Mehr ist das berühmte FOREVER nicht

Wir haben hier erstmals ein Makro eingesetzt. Die Sache ist ganz einfach. Sie schreiben »#define NameDesMakros Inhalt«. Das ist reine Textverarbeitung. In diesem Fall setzt der Compiler für FOREVER – wann immer es im Text auftaucht – einfach *for(;;)* ein. Natürlich läuft diese Schleife auch nicht für immer, sondern nur, solange niemand die [x]-Taste drückt. Die Schreibweise hat sich jedoch eingebürgert.

Den Komma-Operator anwenden

Bisher hatten wir das Komma schon in *printf()* angewandt, um Ausdrücke zu trennen, doch diese Regel ist durchgängiger, als man vielleicht denkt. Was halten Sie von Listing 2.4.9?

```
main()
{
    int i, j;
    i = 10, j = i;
    while(i--, j -= 3, j > 0)
      printf("%4d% 4d\n", i, j);
}
```

Listing 2.4.9: Dürfen es ein paar Ausdrücke mehr sein?

Da, wo ein Ausdruck erlaubt ist, dürfen es auch ein paar mehr sein. Beginnen wir mit dem Schlimmsten, nämlich der while-Zeile. Hier finden Sie drei Ausdrücke durch Kommas getrennt. Die drei Ausdrücke werden der Reihe nach von links nach rechts entwickelt, der letzte bestimmt das Ergebnis, berücksichtigt aber alle vorherigen Ergebnisse. Weil hier als letzter Ausdruck *j > 0* steht, läuft die Schleife nur so lange, wie diese Bedingung wahr ist. Weil vorher mit *j -= 3* die Variable *j* die Werte 7, 4, 1, –2 angenommen hatte, kann die Schleife nur dreimal laufen.

Auch die Zeile *i=10, j=i;* folgt dieser Komma-Regel. Und weil das so schön ist, findet man solche Gebilde hauptsächlich in for-Schleifen, zum Beispiel in dieser Art.

```
for(i=1, j=10; i<10; i++, j++)
  printf("%4d% 4d\n", i,j);
```

Zum Anfang mit continue

In einer Schleife besteht häufig das Problem, daß man bei bestimmten Werten der Laufvariablen nichts tun will. Die einfachste Lösung dafür heißt *continue* (fortsetzen). Schauen wir uns Listing 2.4.10 an.

```
main()
{
    int i;
    for(i=7; i <= 70; i++)
    {
      if ( i % 7 )
        continue;
      printf("%d\n", i);
    }
}
```

Listing 2.4.10: Die Anwendung von continue

Das Programm soll auf eine zugegebenermaßen besonders umständliche Art das kleine Einmaleins mit der Sieben ausgeben. Die for-Schleife läuft aber in Einerschritten von 7 bis 70. Folglich darf alles, was nicht ohne Rest durch 7 teilbar ist, auch nicht gedruckt werden. Genau das prüft der Modulo-Operator. Der Modulo-Operator ist in C das %-Zeichen. *x%y* ergibt den Rest der Division x/y. Ist dieser Rest nicht null, brauchen wir die Zahl nicht. In diesem Fall wirkt *continue*. Es wirkt wie ein Sprung zum Schleifenkopf, wo die nächste Iteration (Erhöhung des Schleifenzählers) ausgeführt wird.

Workshop

Checkliste

1. Das folgende Programm sollte eigentlich nur dann »3 ist gleich 4« drucken, wenn das wahr ist (also nie), doch leider tut es das immer. Wo liegt der Fehler?

```
main()
{
  if (3==4);
    printf("3 ist gleich 4");
}
```

2. Was passiert, wenn man in einer switch-Anweisung nach einem *case* das *break* vergißt?
3. Wie kann man eine Endlosschleife ohne *goto* aufbauen?

Ideen für eigene Übungen

1. Schreiben Sie ein Programm, das in einer for-Schleife die Zahlen von 1 bis 100 mittels *printf()* ausgibt. Nach jeweils fünf Zahlen soll eine neue Zeile begonnen werden.

   Mit Hilfe des Modulo-Operators sähe die Lösung der Aufgabe so aus:

```
main()
{ int i;
  for(i=1;i<=100;i++) {
    printf("%5d",i);
    if ( i % 5 == 0)
      printf("\n");
  }
}
```

   Schreiben Sie aber trotzdem noch ein Programm, das einen Zähler einsetzt.

2.5 Fünfte Sitzung: Ein-/Ausgabe und das Malprogramm

Die Themen dieser Sitzung:

- So entsteht das Malprogramm
- Input und Output
- Maus und Tastatur
- Die IDCMP-Flags
- Messages empfangen und auswerten

Information

Im Laufe der folgenden Sitzungen soll ein Malprogramm erstellt werden. Das Programm wird recht einfach sein und keinen Vergleich mit Deluxe Paint oder anderen Profisystemen aushalten. Das soll es aber auch nicht, es hat einen ganz anderen Zweck. Unser Programm wird Windows, Menüs und verschiedene Requester haben. Sie können damit zeichnen, Text ein- und in verschiedenen Zeichensätzen ausgeben. Sie können Daten auf die Diskette schreiben und von dort lesen. Kurzum: Sie werden ein typisches Amiga-Programm entwickeln, das alles hat, was man meistens braucht. Wenn Sie den Menüs und Requestern andere Texte geben und die zugeordneten Funktionen neu schreiben, können Sie das Malprogramm zu einer Datenbank oder was auch immer umfunktionieren. Sie haben den Rahmen, die ganze Bedienoberfläche und die generelle Abfrage- und Verzweigungstechnik. Sie werden sehen, daß dies einen sehr großen Teil jeder Anwendung ausmacht.

Bevor wir soweit sind, müssen wir allerdings noch ein paar Grundlagen erarbeiten. So ist es sicherlich einsichtig, daß unser Programm auf Mausklicks und Tastendrücke reagieren muß. Es wäre auch ganz schön, wenn wir einen Text in einem Intuition-Window ausgeben könnten (*printf()* geht hier nicht). Also müssen wir uns mit dem Thema Ein- und Ausgabe, Input und Output oder kurz I/O genannt, befassen.

Input und Output via IDCMP

Mit den C-Funktionen wie *scanf()* und *printf()* ist nur ein Input bzw. Output auf Textebene im CLI möglich. Unter Intuition laufen alle Eingaben von der Tastatur oder der Maus über den IDCMP (Intuition Direct Communication Message Port), der Output in Form von Texten und Grafiken wird in das Window »gemalt«. Ein Port ist nichts weiter als eine Datenstruktur (*struct*), in der Intuition Nachrichten hinterlegt, die Sie dann daraus lesen müssen. Außerdem müssen Sie mit *ReplyMsg()* den Erhalt der Nachricht quittieren, so daß eine geordnete Kommunikation stattfindet.

Von Events getrieben

Diese Nachrichten – auch als Briefe vorstellbar – melden Events (Ereignisse). Ein Event ist jede Aktion des Anwenders wie zum Beispiel »Maus bewegt«, »Taste gedrückt« oder »Diskette ausgeworfen«. Ihr Programm tut erst einmal gar nichts, sondern wartet nur auf solche Events. Trifft eines ein, bearbeitet es den Fall und setzt sich dann wieder zur Ruhe, um auf das nächste Event zu warten. Diese Technik nennt man »event driven« oder ereignisgesteuert.

Das IDCMP ist in diesem Sinne der Briefkasten und Intuition der Briefträger, allerdings ein ganz untypischer Postbeamter. Sie können nämlich zum Beispiel sagen: Ich möchte nur über ein Window-Close-Ereignis informiert werden, ich lasse aber andere Ereignisse zu, die dann Intuition selbständig bearbeiten soll. Ein Beispiel dafür: Sie erlauben, daß ein Window verschoben werden darf, aber diese Arbeit selbst lassen Sie Intuition machen.

Das IDCMP besteht aus zwei Message-Ports, nämlich dem *WindowPort* und dem *UserPort*. Unser Briefkasten ist der *UserPort*, den *WindowPort* braucht Intuition für sich selbst. Beide Ports werden automatisch von Intuition angelegt, wenn Sie ein Window öffnen und dabei mindestens ein IDCMP-Flag setzen. Durch das Setzen dieser Flags sagen Sie Intuition, welche Ereignisse es an Ihr Programm melden soll. Heißt Ihre NewWindow-Struktur *nw*, können Sie zum Beispiel mit

```
nw.IDCMPFlags = CLOSEWINDOW | MOUSEMOVE
```

vorgeben, daß Intuition Ihrem Programm nur dann eine Nachricht schicken soll, wenn das Window geschlossen oder die Maus bewegt wird.

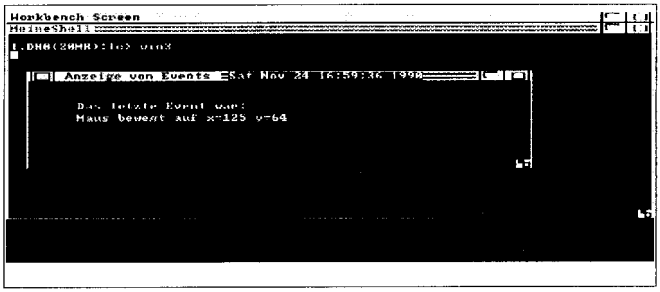


Bild 2.5.1: Das ist unser Ziel

Die IntuiMessage-Struktur

Der Briefkasten, in dem diese Nachrichten ankommen, ist der *UserPort*, und dieser ist ein *struct* vom Typ *IntuiMessage*. Weil wir auf dessen Komponenten immer wieder zugreifen müssen, wollen wir uns das Gebilde laut Listing 2.5.1 einmal Stück für Stück ansehen.

```
struct IntuiMessage
{ struct                Message ExecMessage;
  ULONG                 Class;
  USHORT                Code;
  USHORT                Qualifier;
  APTR                  IAddress;
  SHORT                 MouseX, MouseY;
  ULONG                 Seconds, Micros;
  struct Window         *IDCMPWindow;
  struct IntuiMessage   *SpecialLink;
};
```

Listing 2.5.1: Das Geheimnis des IDCMP (natürlich eine Struktur)

ExecMessage wird von Exec (Kern des Amiga-Betriebssystems) benötigt. Für uns gilt: Das Berühren mit den Pfoten ist verboten! Gleiches gilt für *SpecialLink*.

Class: Die Bits hier entsprechen direkt den IDCMP-Flags, die wir beim Window-Öffnen gesetzt haben, nur können wir hier abfragen, ob das Event eingetreten ist (wie, zeige ich gleich).

Was *Code* für eine Bedeutung hat, hängt vom Wert in *Class* ab. Sagt zum Beispiel *Class*, daß es ein Tastatur-Event ist, so können Sie in *Code* den Tasten-Code lesen, bei einem Menü-Event die Menü-Nummer.

Qualifier: Diese Komponente benötigen wir manchmal, wenn wir die Tastatur abfragen (siehe auch *Code*). *»Qualifier« sind Tasten wie Shift,* die zusammen mit anderen Tasten gedrückt werden.

IAddress: Hier finden Sie die Adressen von Intuition-Objekten (deren Strukturen). Wenn zum Beispiel *class* sagt, daß ein Gadget betätigt wurde, so können Sie hier die Adresse des Gadgets erfahren. Bei User-Gadgets (eigenen Gadgets) müssen Sie hier nachsehen.

MouseX, MouseY enthalten die Maus-Koordinaten (relativ zur linken oberen Ecke des Windows).

Seconds, Micros (Sekunden, Mikrosekunden): Hier steht eine Kopie der Systemzeit. Intuition bedient ein Window (und damit auch diese Daten) aber nur 10- bis 60mal pro Sekunde, die Uhr ist also nicht für Kurzzeitmessungen geeignet.

IDCMPWindow ist die Adresse des Windows für die Messages, also sozusagen die Postanschrift für die Briefe.

Praxis

Mit Listing 2.5.2 kommen wir zu einem Programm, das die Aufgabe hat, Events darzustellen. Zuerst zeigt es ständig in der Titelleiste des Windows das Datum und die Uhrzeit bis hin zur Sekunde an. Im Window selbst steht der Text »Das letzte Event war:« und darunter einer von drei möglichen Texten, nämlich, »Taste .. gedrückt«, »Maus bewegt auf x=.. y=..« und »Größe (des Window) geändert«. Ein Klick auf das Close-Gadget (Schließ-Gadget) beendet das Programm.

```
/* win3.c
   Ein Programm zur Anzeige verschiedener
   Events.
*/
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <time.h>
/* Variable für Windows und Grafik */
struct IntuitionBase *IntuitionBase;
```

Listing 2.5.2: (Fortsetzung folgende Seiten)

```
struct GfxBase *GfxBase;
struct Window *Window;
struct RastPort *rp;
#include "stdwindow.h" /* siehe 3. Sitzung */

/* Variable zur Message-Bearbeitung */
struct IntuiMessage *msg;
ULONG class;
USHORT code;
SHORT mx, my;
char string[81];
struct tm *tp; /* für Datum/Uhrzeit */
time_t  t;

/* ---------------------------------------
   print() gibt den Text in 'string' an der
   Position x,y aus. Vorher wird die Zeile
   durch Überschreiben mit Blanks gelöscht.
   --------------------------------------- */
void print(int x, int y, char *string)
{
 SetAPen(rp, 1L);
 Move(rp, x,y);
 Text(rp,"                                ",30L);
/*        <------ 30 Leerstellen ------>    */
 Move(rp, x,y);
 Text(rp, string, strlen( string) );
}

void _main()
{
 open_libs();

 Window = (struct Window *) open_window(
   20,20,500,100,"Anzeige von Events",

   WINDOWCLOSE |   /* Die Window-Flags */
```

Listing 2.5.2: (Fortsetzung folgende Seiten)

```
    WINDOWSIZING |
    WINDOWDRAG |
    WINDOWDEPTH |
    REPORTMOUSE |

    ACTIVATE,

    CLOSEWINDOW | /* Die IDMCP-Flags */
    NEWSIZE |
    MOUSEMOVE |
    VANILLAKEY |
    INTUITICKS,
    NULL
    );
if (Window == NULL) exit(FALSE);

rp = Window->RPort;
print(50,38,"Das letzte Event war:");

for(;;) /* FOREVER bis CLOSEWINDOW */
{
 Wait(1L << Window->UserPort->mp_SigBit);

 while( msg = (struct IntuiMessage *)
              GetMsg(Window->UserPort) )
 {
   class = msg->Class;
   code = msg->Code;
   mx = msg->MouseX;
   my = msg->MouseY;
   ReplyMsg( msg );
 switch( class)
 {
  case CLOSEWINDOW: close_all();
                    break;
  case NEWSIZE    : print(50,50,"Größe geändert");
                    break;
```

Listing 2.5.2: (Fortsetzung folgende Seite)

```
    case MOUSEMOVE   : sprintf(string,
                         "Maus bewegt auf x=%d y=%d",
                          mx, my);
                       print(50,50,string);
                       break;
    case VANILLAKEY  : sprintf(string,
                          "Taste %c gedrückt",code);
                       print(50,50,string);
                       break;
       case INTUITICKS : time(&t);
                         tp = localtime(&t);
                         Move(rp, 200, 7);
                         Text(rp, asctime(tp),24 );
                         break;
     } /* end switch */
   }   /* end while  */
 }     /* end for    */
}      /* end _main  */
```

Listing 2.5.2: Ein Programm zur Event-Abfrage und Anzeige

Gegenüber dem Listing 2.3.5 aus der dritten Sitzung gibt es nur Erweiterungen, Sie können also darauf aufbauen. Zwei neue Include-Files kommen hinzu. *gfx.h* wird für die Grafikfunktionen gebraucht, die zuständige Library öffnet schon die Funktion *open_libs()* aus dem Include-File *stdwindow.h*. Auch das File kann aus der dritten Sitzung übernommen werden. *time.h* enthält Typen und Funktionen zum Thema Datum/Uhrzeit. Beachten Sie auch, daß ein paar mehr Variablen eingeführt wurden.

Text in einem Window ausgeben

Zu Anfang des Listings gibt es die Funktion *print()*, und das hat Gründe. Unter Intuition können Sie nämlich keinen Text mit *printf()* ausgeben, sondern müssen dafür eine Intuition-Funktion einsetzen, die den Text malt. *printf()* tut das letztlich auch, braucht aber ein CLI-Fenster. Die einfachste Textausgabefunktion unter Intuition heißt schlicht *Text()* mit der Syntax

```
Text(rp, *puffer, anzahl);
```

Darin ist *rp* der Zeiger auf den *RastPort*, **puffer* ein Zeiger auf einen Zeichen-Array mit dem Text und *anzahl* die Anzahl der auszugebenden

Zeichen. Der *RastPort* ist eine Datenstruktur, in der notiert wird, wie momentan gezeichnet wird. Die Vorder- und Hintergrundfarbe, der Zeichenmodus (z.B. überschreibend oder transparent) und andere Informationen sind hier festgehalten. Die meisten Grafikfunktionen benötigen diese Angaben, weshalb man ihnen *rp* übergeben muß. Dieser Zeiger wird mit *rp = Window->RPort;* ermittelt, d.h., in der Window-Struktur gibt es einen Zeiger auf den *RastPort*.

Vor der Textausgabe muß mit *Move(rp, x,y);* der Grafik-Cursor positioniert werden. Ab dieser Stelle wird dann der Text geschrieben. Beachten Sie, daß x und y in Bildpunkten relativ zur linken oberen Ecke des Windows angegeben werden müssen. Vor der Textausgabe sollte man auch noch die Zeichenfarbe setzen. Das geschieht hier mit *SetAPen(rp, 1L);*. *1L* (L für Langzahl) heißt weiß. In der Standardauflösung haben Sie die Farben 0 bis 3, meistens Rot, Weiß, Blau und Schwarz. Alle drei Schritte sind in der Funktion *print()* zusammengefaßt. Dieser Funktion wird *x, y* und ein String mit dem Text übergeben. Beachten Sie: *Text()* erwartet zwar einen char-Array, ich übergebe aber einen String. Das hat den Vorteil, daß man mit *strlen()* die Anzahl der Zeichen ermitteln kann (das Null-Byte wird dabei nicht gezählt) und so die Länge nicht immer abzählen muß. Damit reduziert sich schließlich unsere ganze Textausgabe auf zum Beispiel »print(100,50,"Hallo");«.

Window-Flags und IDCMP-Flags setzen

Sie wissen noch aus der dritten Sitzung: Ein Window wird geöffnet, indem man die NewWindow-Struktur mit Daten versorgt und damit *OpenWindow()* aufruft. Die Details dazu haben wir in der Funktion *open_window()* versteckt, die wiederum im Include-File *stdwindow.h* steht. Hier also rufen wir nur noch *open_window()* auf, und das unter anderem mit diesen Window-Flags:

Window-Flag	Wirkung
WINDOWCLOSE	Window bekommt Close-Gadget
WINDOWSIZING	Window bekommt Größen-Gadget
WINDOWDRAG	Window kann verschoben werden
WINDOWDEPTH	Window bekommt Vorder-/Hinter-Gadget
REPORTMOUSE	Window meldet Maus-Bewegungen
ACTIVATE	Window ist sofort aktiv

Tabelle 2.5.1: Window-Flags

Beachten Sie, daß das alles Bit-Werte (Zahlen) sind, die mittels der Oder-Verknüpfung (|) zu einem einzigen Ausdruck zusammengefaßt werden. Das nächste Argument sind die IDCMP-Flags, gleichfalls Bitwerte, die auch »verodert« werden.

IDCMP-Flag	Intuition bringt Message, wenn
CLOSEWINDOW	Window geschlossen wurde
NEWSIZE	Größe geändert wurde
MOUSEMOVE	Maus bewegt wurde
VANILLAKEY	Taste gedrückt wurde
INTUITICKS	Uhrzeit gesendet wurde

Tabelle 2.5.2: IDCMP-Flags

Erklärungsbedürftig ist VANILLAKEY. Alternativ dazu (nicht gleichzeitig) können Sie RAWKEY schreiben. Wenn dieses Bit gesetzt ist, sendet Intuition den »rohen« Tasten-Code als eine Zahl zwischen 0 und 255. Diesen Code müßten Sie erst in den ASCII-Code umrechnen. Nur mit letzterem bringen die C-Funktionen vernünftige Texte auf den Schirm. Um die einfachere automatische Umrechnung zu aktivieren, müssen Sie das Bit VANILLAKEY setzen. Merke: Im Amerikanischen ist Vanilla ein Ausdruck für etwas, das schön einfach ist. Die Sache hat einen Haken: VANILLAKEY meldet alle Sondertasten, wie z.B. die Funktionstasten nicht.

Wir warten auf Post von Intuition

Das Window ist geöffnet, wir ermitteln noch mit *rp = Window->RPort;* den *RastPort*, geben mittels »print(50,38,"Das letzte Event war:");« diesen Text aus und gehen dann mit *for(;;)* in eine Endlosschleife. Diese beginnt mit der tollen Zeile

```
Wait(1L << Window->UserPort->mp_SigBit);
```

Im Klartext heißt das etwa: Das Programm soll warten, bis der Wecker klingelt. Dabei könnte ich es bewenden lassen, doch Sie wollen ja C lernen, also wie funktioniert das?

Damit das Programm nicht ständig abfragen muß, ob ein bestimmtes Ereignis eingetreten ist, kann es sagen: »Ich gehe jetzt schlafen, wecke mich wieder, wenn Post in meinem Briefkasten ist.« Man könnte auch

sagen »der Briefträger soll klingeln«. Dazu muß das Programm die Funktion *Wait()* aufrufen, und zwar mit einer Bit-Maske (letztlich einer Zahl) als Argument. In dieser Maske muß genau das Bit gesetzt sein, das meinem Window entspricht. Ein Programm kann mehrere Windows verwalten, dann hat jedes ein Bit. Nun weiß ich aber nicht, welches Bit mein Window ist, ich weiß aber, wo das steht, nämlich in *mp_SigBit* (Signal-Bit).

Der Weg dahin ist lang. Mit dem Öffnen des Windows hatten wir den Zeiger *Window* auf die Window-Struktur erhalten. In dieser Window-Struktur gibt es einen Zeiger auf die UserPort-Struktur und in dieser einen Zeiger auf *mp_SigBit*. Nun wissen Sie noch, daß man bei Zeigern auf Strukturen immer -> anstatt des Punktes schreiben muß, also ist der Weg *Window->UserPort->mp_SigBit* schon klar.

Bliebe noch das *1L* << am Anfang des Ausdrucks. Das Ergebnis von *Window->UserPort->mp_SigBit* ist eine Zahl, zum Beispiel 3. Damit reduziert sich der ganze Ausdruck auf

```
Wait( 1L << 3 );
```

Die Zeichenfolge << ist der Links-Shift-Operator. *1L* heißt die Zahl 1 als Langzahl. Die 3 ist aber eine Bit-Nummer und *Wait()* will eine Maske sehen, in der Bit 3 gesetzt ist. Daher mache ich folgendes: Ich nehme die Zahl 1, also binär 00000001, wo Bit 0 gesetzt ist. Dann schiebe ich die 1 um 3 Bit nach links und habe somit 00001000, was übrigens der Zahl 8 entspricht.

Es gibt übrigens auch fiese Programmierer, die ohne *Wait()* arbeiten und statt dessen schreiben

```
while( msg = GetMsg(Window->UserPort) == 0 )
 ;
```

Im Klartext: Solange keine Message anliegt (== 0) frage immer wieder. Das ist die sogenannte Polling-Methode. Aus CPU-Sicht vergehen aber Ewigkeiten, bis mal wirklich eine Message eintrifft. Folglich ist die Antwort vielleicht 9999mal nein und dann einmal ja. Natürlich kostet diese Fragerei auch Rechnerzeit, kostbare Zeit, die anderen Tasks fehlt. In einem Multitasking-System – und das haben wir beim Amiga – ist das ein grober Fehler.

Wir werten die Post aus

Wie auch immer, *Wait()* bewirkt also, daß unser Programm an dieser Stelle so lange wartet, bis die Post klingelt. Ist das der Fall, müssen wir zum Briefkasten gehen und die Briefe dort herausholen, und zwar solange (while) das Erfolg hat. Wir nehmen aber nicht den Stapel Briefe in die Hand, sondern ziehen einen aus dem Kasten, lesen ihn und schicken dann Intuition sofort eine Quittung der Art »Brief erhalten«. Den Inhalt des Briefs hatten wir uns notiert und bearbeiten jetzt den Fall. Dann ziehen wir den nächsten Brief und verfahren entsprechend. Das geht so lange bis der Kasten leer ist. An dieser Stelle legen wir uns wieder hin und warten darauf, daß der Briefträger klingelt. Wenn der das nicht bis zu 60-mal pro Sekunde machen würde, hätten wir einen ganz tollen Job. Das Spielchen geht so lange, bis da ein Brief erscheint mit der Meldung, daß der Anwender das Close-Gadget angeklickt hat. An dieser Stelle schließen wir mit *close_all()* alle Libraries und das Window und brechen das Programm ab.

Dieser Ablauf sieht in C-ähnlicher Schreibweise etwa so aus:

```
for(;;) /* an sich endlos */
 {
   Warte bis Briefträger klingelt;

  while( Brief aus Kasten nehmen == erfolgreich )
   {
     class = Art der Nachricht;
     Quittiere Nachricht;

   switch( class)
     {
       case CLOSEWINDOW: Alles schließen und Ende;
                         break;
       case NEWSIZE    : Behandle "Größe geändert";
                         break;
         case u.s.w.
       }
    }
 }
```

In echter C-Schreibweise wird aus dem obigen »Listing« diese Form:

```
for(;;) /* FOREVER bis CLOSEWINDOW */
 {
  Wait(1L << Window->UserPort->mp_SigBit);

  while( msg = GetMsg(Window->UserPort) )
  {     .
    class = msg->Class;

    ReplyMsg(Msg);

    switch( class)
    {
      case CLOSEWINDOW: close_all();
                        break;
      case NEWSIZE : print(50,50,"Größe geändert");
                        break;
    }
  }
 }
```

Wenn Sie das mit Listing 2.5.2 vergleichen, werden Sie es wiedererkennen, es sind nur ein paar Details mehr, zum Beispiel diese:

```
while( msg = (struct IntuiMessage *)
              GetMsg(Window->UserPort) )
  {
    class = msg->Class;
    code = msg->Code;
    mx = msg->MouseX;
    my = msg->MouseY;
    ReplyMsg( msg );
```

Sobald die Message gelesen wurde, fangen wir an, alles, was uns interessiert, aus der msg-Struktur in andere Variable umzuspeichern, zum Beispiel mit *class = msg->Class;*. Der Grund ist: Sobald wir mit *ReplyMsg(msg);* den Empfang quittiert haben, sind die Daten in msg nicht mehr gültig. Intuition kann sofort danach neue Daten eintragen.

Ihnen ist angenehm aufgefallen, daß in *msg->class* bzw. in *class* dieselben Bezeichner wie in den IDCMP-Flags stehen? Das ist korrekt. Mit

diesen Flags sagen wir Intuition, was es uns melden soll, mit denselben Flags fragen wir ab, welche Nachricht anliegt.

Wir tricksen mit sprintf();

Wie eingangs erwähnt, können wir mit der Funktion *Text()* – versteckt in der Funktion *print()* – nur Texte drucken. Das ist ausgesprochen ungünstig, wenn wir Zahlen ausgeben müssen, die in Variablen stehen. Diese Zahlen müssen nämlich vom rechnerinternen Binärformat (0101010101) in für uns lesbare Zeichenfolgen gewandelt werden. Genau diesen Job haben wir bisher *printf()* erledigen lassen, und das ist jetzt verboten. Die Lösung heißt *sprintf()* (string-printf). Diese Funktion arbeitet genauso wie *printf()*, aber mit einem Unterschied. Die Zeichen werden nicht auf den Schirm, sondern in einen String geschrieben. Ein Beispiel: In *code* steht der ASCII-Code der zuletzt gedrückten Taste. Dieser Code soll als Zeichen ausgegeben werden. Mit *printf()* würde man dann schreiben:

```
printf("Taste %c gedrückt",code);
```

Der Unterschied zu *sprintf()* ist recht gering. Man muß nur noch den String angeben, in den das Ergebnis geschrieben werden soll. Dafür haben wir im Programm schon mit *char string[81];* eine Variable deklariert und schreiben daher nur noch

```
sprintf(string, "Taste %c gedrückt",code);
```

Nun haben wir wieder unseren String und können ihn mit *print(50,50, string);* ausgeben.

Wir lassen eine Uhr laufen

Im case-Selektor für INTUITICK steht diese geheimnisvolle Kommando-Folge:

```
time(&t);
tp = localtime(&t);
Move(rp, 200, 7);
Text(rp, asctime(tp),24 );
```

Sinn der Übung: Immer wenn ein INTUITICK-Event eintrifft, sollen das Datum und die Uhrzeit angezeigt werden. Einen »Intuition-Tick« von der Systemuhr gibt es ungünstigstenfalls zehnmal pro Sekunde, was für eine Uhr mit Sekundenanzeige reicht. Würden wir die Routine in die Hauptschleife legen, würde sie zu oft aufgerufen, und der Text würde flackern.

Das Event INTUITICK ist aber nur der Anlaß, die Routine zu starten. Jetzt kommt die Arbeit. Die Zeit steht in einer Systemvariablen vom Typ *long*. Aus geheimnisvollen Gründen hat man aber dafür im Include-File *time.h* den Typ *time_t* definiert. Wie auch immer, in dieser Langzahl steht die Zeit in Sekunden ab Mitternacht vom 1. Januar 1978. Unterschätzen Sie das nicht, wegen des Long-Wertes reicht das noch für über 100 Jahre.

Die Funktion *time(&t)* liest diesen Sekundenwert aus und speichert ihn in der Variablen *t*. Der nächste Aufruf, nämlich *tp = localtime(&t);* rechnet die Sekunden um und speichert das Ergebnis in einer Struktur, auf die *tp* zeigt. Das *struct besteht aus lauter int-Komponenten* für Sekunden, Minuten, Stunden, Tag, Monat und Jahr. Die Funktion *asctime()* schließlich wandelt diesen *struct* in einen String um, wobei für die Monatsnummer sogar noch Text (die ersten Buchstaben) eingesetzt wird. Das Ergebnis ist ein String von 26 Zeichen, wir drucken aber nur 24, weil wir das »\n\0« am Ende nicht brauchen. Bliebe noch zu erwähnen, daß ich den Zwischenschritt *tp = localtime(&t);* nur der Lesbarkeit wegen eingesetzt habe. Man kann diese Zeile auch weglassen und dann schreiben:

```
Text(rp, asctime( localtime(&t)  ),24 );
```

Sie wissen noch: Anstatt einer Variablen, die ein Funktionsergebnis hält, kann man auch gleich den Funktionsaufruf einsetzen.

Workshop

Checkliste

1. Warum sollen Sie die Polling-Methode vermeiden und statt dessen *Wait()* einsetzen?
2. Was ist der Unterschied zwischen RAWKEY und VANILLAKEY?
3. Warum muß man vor *ReplyMsg()* die noch benötigten Werte im Message-Port sichern?

Ideen für eigene Übungen

1. Erweitern Sie mit Hilfe der IDCMP-Flags (im open_window()-Aufruf) DISKREMOVED und DISKINSERTED sowie neuer case-Selektoren das Programm so, daß im Falle eines Diskettenauswurfs bzw. -einschubs die Meldung »Diskette ausgeworfen« bzw. »Diskette eingeschoben« erscheint.

   Kompilieren, linken und testen Sie das Programm.

2. Ändern Sie im open_window()-Aufruf und im case-Selektor VANILLAKEY in RAWKEY. Stellen Sie in *sprintf()* von Zeichenausgabe (%c) auf Zahlenausgabe (%d) um. Beobachten Sie, welche Tasten welche Codes haben. Ändern Sie dann das Programm so, daß es auch mit einem Druck auf die F1-Taste beendet werden kann.

   Kompilieren, linken und testen Sie das Programm.

2.6 Sechste Sitzung: Unser Window bekommt Menüs

Die Themen dieser Sitzung:

- Aufbau eines Menüs
- Menü-Strukturen und Flags
- Programmieren von Menüs
- Der Menü-Code
- Makros

Information

Zuerst besteht ein Menü aus der Menüleiste (Titelzeile), in Intuition-Schreibweise *MenuStrip* (Menü-Streifen). Jeder einzelne Titel innerhalb des Streifens/der Leiste heißt *Menü* (Menu). Eine Menüleiste gehört immer zu einem Window. Das Menü kann nur angewählt werden, wenn das Window aktiv ist. Wurde das Menü angewählt, erscheint die Menüleiste. Nun müssen Sie mit der Maus ein Menü herunterrollen. Die dann sichtbaren Unterpunkte nennt man *Items* oder Menüpunkte.

Die Auswahl eines Items kann aber auch zur Folge haben, daß plötzlich weitere Items sichtbar werden. Das sind dann die sogenannten Sub-Items (Untermenüpunkte). Ein Sub-Item kann durchaus wieder ein Sub-Sub-Item produzieren. Theoretisch kann das noch weitergehen, praktisch wird damit die Bedienung zu kompliziert. Meistens sind die Items Text. Zum Beispiel werden Sie unter dem Menü-Titel »Projekt« üblicherweise den Item-Text »Laden« finden. Es hindert Sie aber niemand daran, anstatt dieses Textes eine Grafik abzubilden. Intuition unterstützt direkt und sehr gut Grafiken in Menüs.

Oft ist es zu umständlich, erst das Menü und dann ein Item anzuwählen. Als Lösung des Problems bietet Intuition sogenannte Command-Keys (Kommando-Tasten). Eine Kommando-Taste ist eine Taste der Tastatur, die direkt erreichbar ist (zum Beispiel nicht zusammen mit der [Alt]-Taste). Wird die [rechte Amiga]-Taste gleichzeitig mit diesem Command-

Key gedrückt, hat das dieselbe Wirkung, wie die Auswahl des zugeordneten Items, und zwar automatisch.

☞ Geben Sie niemals kritischen Operationen einen Command-Key. Diese Kommandos werden nämlich direkt und sofort ausgeführt. Bei einem Menü-Item können Sie die Maus immer noch wegziehen, aber eine Kommando-Taste kann unbeabsichtigt gedrückt worden sein.

Aufbau eines Menüs

In dieser Sitzung werden wir einen MenuStrip aufbauen, der zwei Titel (Menüs) hat, nämlich »Projekt« und »Farben«. Das Projekt-Menü hat die Items »Neu«, »Laden«, »Sichern« und »Ende«. Letzteres hat die beiden Sub-Items »Wirklich« und »Lieber doch nicht«. Für »Wirklich« gibt es den Command-Key »`[Amiga] + [x]`«. Das Farben-Menü hat die Items »Rot«, »Grün« und »Blau«.

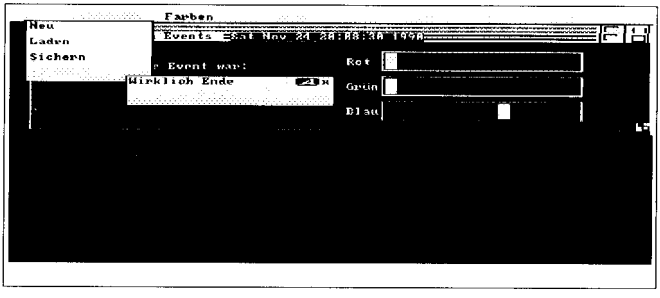


Bild 2.6.1: Das ist unser Ziel

Wie Screens und Windows sind auch Menüs Strukturen, die unter anderem einen Zeiger auf das nächste Menü enthalten. Hier gilt:

```
Projekt --> Farben --> NULL
```

In jeder Menü-Struktur gibt es (unter anderem) Zeiger auf die Menü-Titel. Aus diesen wird der MenuStrip aufgebaut. In jedem Menü gibt es einen Zeiger auf die Item-Liste. Hier gelten dann die Folgen:

```
Neu --> Laden --> Sichern Ende --> NULL
Rot --> Grün --> Blau --> NULL
```

Auch die Items sind Strukturen. Jedes Item hat einen Zeiger auf die Sub-Item-Liste. Ist dieser Zeiger nicht NULL (d.h. es gibt Sub-Items), gilt wieder:

```
Wirklich  --> Lieber doch nicht --> NULL
```

Innerhalb der Items und Sub-Items gibt es einen Zeiger, der auf noch eine Struktur zeigt. Diese ist entweder eine Textstruktur (kein einfacher String) für den Item-Text oder eine Image- Struktur, wenn Grafik anstatt Text dargestellt werden soll. Daß die Textstruktur noch einen Zeiger hat, der dann auf den Text-String zeigt, sei nur noch draufgesetzt. Auf jeden Fall sehen Sie schon, wir werden wieder fröhlich »zeigern« müssen. Die 14 wichtigsten Zeiger stellt Bild 2.6.2 dar.

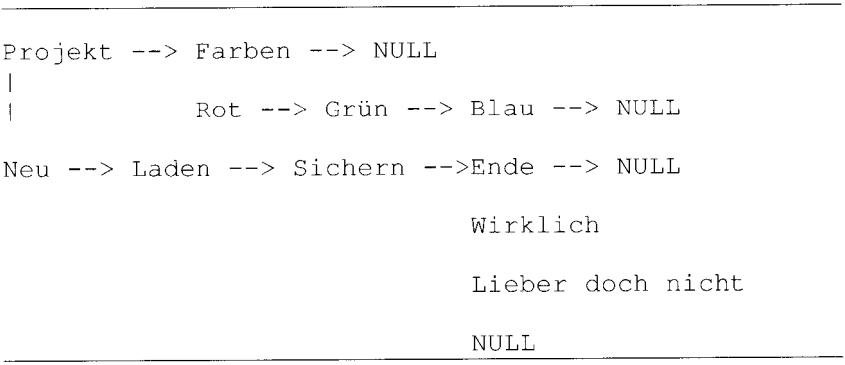


Bild 2.6.2: Die wichtigsten Zeiger in einem Menü-System

Praxis

Bild 2.6.1 ist unser Ziel, doch nun gilt es, das in die Praxis umzusetzen. Für *Projekt* und *Farben* (in Bild 2.6.1) müssen wir zwei structs vom Typ *Menu* einsetzen. Im Include-File *intuition.h* ist *Menu* gemäß Listing 2.6.1 definiert.

```
struct Menu
{
 struct Menu *NextMenu;   /*  -> nächstes Menu     */
 SHORT LeftEdge, TopEdge;/* linke obere Ecke       */
 SHORT Width, Height; /* Breite und Höhe der Box */
 USHORT Flags;            /* Flags (siehe unten)  */
 BYTE *MenuName;          /* --> Titel-Text       */
 struct MenuItem *FirstItem;    /* --> Item-Liste */
 /* Nur für Intuition intern: */
 SHORT JazzX, JazzY, BeatX, BeatY; /* Intui.-intern */
};
```

Listing 2.6.1: Die Menü-Struktur

Menüs gestalten

Mit *LeftEdge* geben Sie die Lage des Titels vor. Für *TopEdge* können Sie zur Zeit schreiben, was Sie wollen, Intuition nimmt immer die obere Schirmkante. Gleiches gilt für die Höhe *Height*, wofür Intuition immer die Höhe der Titelleiste des Screens einsetzt.

Dem Menü selbst können Sie nur ein Flag mitgeben (oder auch nicht, dann 0). Ist MENUENABLED gesetzt, ist das Menü wählbar, sonst ist es deaktiviert und wird grau gezeichnet (inklusive aller Items). Diese Zustände können durch die Funktionen *MenuOn()* und *MenuOff()* nachträglich geändert werden.

**MenuName* ist ein Zeiger auf einen String. Hier reicht es, den Namen direkt in Anführungszeichen einzutragen.

**FirstItem* zeigt auf das erste Item, und damit wären wir bei der nächsten Struktur. Die Item-Struktur gemäß Listing 2.6.2 gilt gleichermaßen für Items und Sub-Items. Alles außer *Projekt* und *Farben* in Bild 2.6.1 muß vom Typ *MenuItem* sein.

```
struct MenuItem
{
 struct MenuItem *NextItem; /* --> nächstes Item */
 SHORT LeftEdge, TopEdge;   /* linke obere Ecke  */
 SHORT Width, Height; /* Breite und Höhe der Box */
 USHORT Flags;              /* Flags (siehe unten) */
 LONG MutualExclude;  /* Gegenseitiger Ausschluß */
 APTR ItemFill;     /* Zeiger auf Text oder Image */
 APTR SelectFill;       /* Text/Image bei Auswahl */
 BYTE Command;                        /* Command-Key */
 struct MenuItem *SubItem; /* --> Sub-Item-Liste */
 USHORT NextSelect;           /* Für Mehrfachwahl */
};
```

Listing 2.6.2: Die Item-Struktur (gilt auch für Sub-Items)

Solange *NextItem* nicht NULL ist, setzt sich die Liste fort. Mehr als 32 Items sind aber nicht erlaubt. Die Lage (*LeftEdge, TopEdge*) gilt relativ zur Lage des Menüs. Die Breite (*Width*) und die Höhe (*Height*) sollten mindestens die Maße des Texts oder der Grafik (Image) abdecken. *NextSelect* berücksichtigt die sogenannte Mehrfach-Anwahl, die wir hier nicht berücksichtigen. Flags gibt es wieder genügend, doch wir benutzen nur zwei.

Flags setzen

Ist das Flag ITEMENABLED gesetzt, ist das Item wählbar. Setzen Sie hier 0 ein, werden auch alle Sub-Items (wenn vorhanden) deaktiviert. Wenn Sie im Command-Feld einen Buchstaben für einen Command-Key eingetragen haben, müssen Sie auch dieses COMMSEQ-Flag setzen. Intuition wird dann [Amiga] [A] mit diesen Buchstaben neben den Text schreiben. Geben Sie dann *width* breiter als den Text an, damit diese beiden Zeichen noch hineinpassen! ITEMTEXT müssen Sie setzen, wenn das Item Text sein soll, sonst müßte *ItemFill* auf ein Grakik-Image zeigen. Außerdem setzen wir noch HICOMP, womit das Item bei der Anwahl invertiert wird. Wenn Sie statt dessen lieber einen Rahmen um das Item gezeichnet haben wollen, setzen Sie HIBOX.

Menü-Strukturen initialisieren

In Bild 2.6.2 gibt es zwei Menü-Strukturen und neun MenuItem-Strukturen. Also müssen wir zwei bzw. neun Variablen dieses Typs anlegen und diese mit Daten füttern, etwas fachlicher gesagt, initialisieren. Das könnte so aussehen:

```
struct Menu Projekt =
 {
 &Farben,          /* Zeiger auf Nachfolger */
 10, 0, 120, 10,   /* Lage und Maße der Box */
 MENUENABLED,      /* Flags */
 " Projekt ",      /* Titel */
 &Neu              /* Zeiger auf Item-Liste */
 };
```

Damit würde man normalerweise anfangen, doch das klappt nicht. Wenn ich *&Farben* schreibe, muß *Farben* schon existieren. In *Farben* gibt es aber den Zeiger auf das Item *Rot*, also muß das noch davorstehen. In *Rot* gibt es den Zeiger auf *Grün*, das muß somit noch weiter vorne stehen. Schließlich gibt es in *Rot* noch den Zeiger auf *TextVonRot*, dieser muß also zuerst deklariert sein. Langer Rede kurzer Sinn: Die Reihenfolge im Listing entspricht nicht mehr der logischen Reihenfolge, ein immer großer Nachteil. Problem zwei: Sie müssen bei den Datentypen höllisch aufpassen. Irren Sie sich da einmal, gerät die ganze Kette durcheinander, der echte Fehlerort ist dann garantiert ein anderer, als der, den der Compiler meldet.

Wenn ich hingegen lauter Variable anlegen, dann nach dem Motto:

```
struct Menu Projekt;
Projekt.Menu      = &Farben:
Projekt.LeftEdge  = 10;
Projekt.TopEdge   = 0;
Projekt.Width     = 120;
Projekt.Heigh     = 10;
```

So kann ich diesen Fehler nicht mehr machen, doch wäre das eine endlose Tipperei, wenn ich diesen Job nicht in Funktionen verlagert hätte. Die Funktionen decken nicht alle Sonderfälle ab, aber wie Sie gleich am Beispiel von »Ende« noch sehen werden, kann man ja struct-Komponenten auch nachträglich andere Werte zuweisen, es sind schließlich Variable. Die Variablennamen selbst sind auch so ein Problem. In großen Programmen mit zum Beispiel 10 Menüs und 100 Items sind das 110 Namen, die schwer zu überblicken und zu handhaben sind. Wir nehmen deshalb Arrays, und zwar

```
struct Menu       menu[2]   /* für die Menüs      */
struct MenuItem   item[9];  /* für die Items      */
struct IntuiText  itext[9]; /* für die Item-Texte */
```

Und damit wären wir schon mitten im Listing 2.6.3. Speichern Sie diese Datei als *menu.h*. Wir werden sie dann später in das Programm aus der fünften Sitzung (Listing 2.5.2) »includen« und müssen dann dort nur noch ganz wenig hinzubringen.

```
/* menu.h

   Alle Funktionen zum Anlegen und zur Abfrage
   von Menüs. (c) 1990 Peter Wollschlaeger
*/

struct Menu       menu[2]   /* für die Menüs  */
struct MenuItem   item[9];  /* für die Items  */
struct IntuiText  itext[9]; /* für Item-Texte */
/* Konstanten für die spätere
   Menüabfrage
```

Listing 2.6.3: (Fortsetzung folgende Seiten)

```
*/
#define PROJEKT  0
#define FARBEN   1

#define NEU      0
#define LADEN    1
#define SICHERN  2
#define ENDE     3

#define WIRKLICH 0
#define NEIN     1

#define ROT      0
#define GRUEN    1
#define BLAU     2


/* Text in ein IntuiText-struct
   eintragen
*/
void make_text(struct IntuiText *name, char *text,
               SHORT left, SHORT top)

{
  name->FrontPen  = 0;
  name->BackPen   = 1;
  name->DrawMode  = JAM1;
  name->LeftEdge  = left;
  name->TopEdge   = top;
  name->ITextFont = 0;
  name->IText = (UBYTE *) text;
  name->NextText = NULL;
}

/* Daten in ein MenuItem-struct eintragen
*/
void make_item( char *itemtext,
```

Listing 2.6.3: (Fortsetzung folgende Seiten)

```
                    struct IntuiText *name,
                    struct MenuItem *item,
                    struct MenuItem *next,
                    USHORT left,USHORT top,
                    USHORT width,USHORT height,
                    ULONG flags )
{

  item->NextItem = next;
  item->LeftEdge = left;
  item->TopEdge  = top;
  item->Width    = width;
  item->Height   = height;
  item->Flags    = flags |
                   ITEMTEXT | HIGHCOMP;
  item->MutualExclude = NULL;

  make_text( name, itemtext, 0,0 );
  item->ItemFill = (APTR) name;

  item->SelectFill = NULL;
  item->Command    = NULL;
  item->SubItem    = NULL;
}

/* Daten in einen Menu-struct eintragen
*/
void make_menu(struct Menu *name, char *titel,
               struct Menu *next,
               struct MenuItem *first,
               USHORT left,USHORT top,
               USHORT width,USHORT height,
               ULONG flags )
{
  name->NextMenu = next;
  name->LeftEdge = left;
  name->TopEdge = top;
```

Listing 2.6.3: (Fortsetzung folgende Seiten)

```
  name->Width = width;
  name->Height = height;
  name->Flags = flags;
  name->MenuName = (BYTE *) titel;
  name->FirstItem = first;
}

/* Das ganze Menü-System aufbauen
*/
void MakeTheMenu()
{

/*----------- Menü 0 ---------------*/

make_item( "Neu",       /* Text des Items          */
         &itext[0],   /* Adresse IntuiText-struct */
         &item[0],    /* Adresse der Item-struct */
         &item[1],    /* Adresse next Item        */
         5,0,         /* linke obere Ecke         */
         120,11,      /* Breite und Höhe          */
         ITEMENABLED /* Flags des Items          */
         );

make_item("Laden",    &itext[1], &item[1], &item[2],
                      5,16,120,11, ITEMENABLED );

make_item("Sichern", &itext[2], &item[2], &item[3],
                      5,32,120,11, ITEMENABLED );

make_item("Ende",     &itext[3], &item[3], NULL,
                      5,48,120,11, ITEMENABLED );

make_item("Wirklich Ende",
                      &itext[4], &item[4], &item[5],
                      100,8,200,11,
                      ITEMENABLED | COMMSEQ );
```

Listing 2.6.3: (Fortsetzung folgende Seiten)

```
make_item("Lieber doch nicht",
                    &itext[5], &item[5], NULL,
                    100,24,200,11, ITEMENABLED );

item[3].SubItem = &item[4]; /* Sub-Item nachtragen */
item[4].Command =  'x';     /* Dito Cmd-Key       */

make_menu(&menu[0]," Projekt ", &menu[1], &item[0],
                    10,0,120,10, MENUENABLED );

/*-------- Menu 1 --------*/

make_item("Rot", &itext[6], &item[6], &item[7],
                 5,0,100,11, ITEMENABLED );

make_item("Grün", &itext[7],&item[7], &item[8],
                  5,16,100,11,ITEMENABLED );

make_item("Blau", &itext[8], &item[8], NULL,
                  5,32,100,11,ITEMENABLED );

make_menu(&menu[1],"Farben", NULL, &item[6],
                   150,0,100,10, MENUENABLED );
}

/* MENUPICK-Message auswerten und zugehörige
   Aktionen ausführen
*/
void do_menu()
{
  switch(MENUNUM(code)) /* Switch Titel */
  {
    case PROJEKT:
         switch( ITEMNUM(code) ) /* Switch Items */
              {
                 case NEU:   print(50,50, "Neu");
                 break;
```

Listing 2.6.3: (Fortsetzung folgende Seite)

```
                case LADEN: print(50,50, "Laden");
                break;
                case SICHERN: print(50,50,
                                        "Sichern");
                break;
                case ENDE: switch( SUBNUM(code) )
                     {
                       case WIRKLICH:
                            ClearMenuStrip(Window);
                            close_all();
                            break;
                       case NEIN: print(50,50,
                             "Lieber doch nicht");
                             break;
                     }
                     break;
           }
           break;

    case  FARBEN: switch( ITEMNUM(code) )
           {
             case ROT: print(50,50, "Rot");
             break;

             case GRUEN:print(50,50, "Grün");
             break;

             case BLAU:print(50,50, "Blau");
              break;
           }
           break;
    }
}
```

Listing 2.6.3: menu.h hat alles, was man für Menüs braucht

Für Sie ist die Funktion *make_item()* die wichtigste, die wie folgt aufgerufen wird:

```
make_item( "Neu",       /* Text des Items          */
          &itext[0],  /* Adresse IntuiText-struct */
          &item[0],   /* Adresse der Item-struct  */
          &item[1],   /* Adresse next Item        */
          5,0,        /* linke obere Ecke         */
          120,11,     /* Breite und Höhe          */
          ITEMENABLED /* Flags des Items          */
          );
```

Die Kommentare sind etwas kurz, deshalb noch diese Hinweise: »Neu« ist der Text des Items. Dieser wird aber nicht in der Item-Struktur abgelegt, sondern in einer Struktur vom Typ *IntuiText*. Diese ist ein Element des Arrays *itext[9]*, hier *itext[0]*. Die Funktion erhält mit *&itext[0]* einen Zeiger darauf. Deshalb kann sie mit

```
make_text(name, itemtext);
```

die Funktion *make_text()* aufrufen. Diese wiederum trägt in die mit *name* bezeichnete IntuiText-Struktur den Text ein. Ist das geschehen, kann ich mit

```
item->ItemFill = (APTR) name;
```

in der MenuItem-Struktur den Zeiger auf diese IntuiText-Struktur eintragen. APTR (a-pointer) ist ein allgemeiner Zeigertyp, der immer dann verwendet wird, wenn ein Zeiger auf unterschiedliche Objekte zeigen soll. *item->ItemFill* kann nämlich auch auf eine Image-Struktur zeigen, wenn das Item eine Grafik sein soll.

Nun zu diesen beiden Zeilen:

```
item[3].SubItem = &item[4]; /* Sub-Item nachtragen */
item[4].Command =  'x';     /* Dito Cms-Key        */
```

Unsere Funktion *make_item()* hat keinen Parameter für Sub-Items, sondern trägt dafür immer NULL ein. Bei *item[3]* ist das falsch, weshalb das mit der ersten Zeile korrigiert wird. Dito wird auf diese Art für *item[4]* der Command-Key 'x' nachgetragen.

Die Menüs erweitern

Wenn Sie die Menüs erweitern wollen, ist das kein Problem. Schauen wir uns dafür das Farben-Menü an. Nehmen wir an, Sie wollen ein Item mit dem Text »Gelb« hinzubringen. Nun nehmen Sie das letzte Item (Blau) und duplizieren es. Im Text steht dann:

```
make_item("Blau", &itext[8], &item[8], NULL,
                  5,32,100,11,ITEMENABLED );

make_item("Blau", &itext[8], &item[8], NULL,
                  5,32,100,11,ITEMENABLED );
```

In der ersten Zeile steht noch NULL (kein Nachfolger), das ändern Sie in

```
make_item("Blau", &itext[8], &item[8], &item[9],
```

Nun ist der neue make_item-Aufruf so anzupassen.

```
make_item("Gelb", &itext[9], &item[9], NULL,
                  5,48,100,11,ITEMENABLED );
```

Wir haben den Text geändert, die Elemente in [9] und die Y-Position von 32 in 48. Vergessen Sie jetzt nur nicht, auch die Arrays

```
struct MenuItem  item[9];  /* für die Items  */
struct IntuiText itext[9]; /* für Item-Texte */
```

von »9« in »10« zu ändern. Wenn Sie ein neues Menü anlegen wollen, duplizieren Sie zuerst die Items wie gehabt. Ihr erstes Item müßte dann *item[10]* heißen und Ihr Menü *menu[2].* Nun duplizieren Sie einen make_menu()-Aufruf und ändern den in

```
make_menu(&menu[2],"Figuren", NULL, &item[10],
                   300,0,100,10, MENUENABLED );
```

Auch der menu-Array muß dann um 1 erhöht werden.

Menüs abfragen mit Makros

Nun kommen wir zu der Funktion *do_menu(),* die für die Abfrage der Menüs und die zugehörigen Aktionen zuständig ist. Wenn der Anwender einen Menüpunkt ausgewählt hat, steht im Code-Feld der *IntuiMessage* ein 16-Bit-Wort, in dem codiert ist, welches Menü, welches Item und welches Sub-Item selektiert wurde. So ein Menü-Wort teilt sich so auf:

```
s s s s s i i i i i i m m m m m
```

Das heißt:

- Bit 0 bis 4: Menü
- Bit 5 bis 10: Item
- Bit 11 bis 15: Sub-Item

Sehr schön, doch wie isoliert man nun die Einzelinformationen? Nun, Leser, die Freude am »Bitschieben« haben, können jetzt voll zuschlagen. Ich empfehle aber dringend, sich beim Programmieren auf die wichtigen Dinge zu konzentrieren, und diese Aufgabe mittels Makros zu erledigen. Folgende Makros sind in *intuition.h* schon eingebaut:

```
#define MENUNUM(n) (n & 0x1F)
#define ITEMNUM(n) ((n » 5) & 0x003F)
#define SUBNUM(n) ((n » 11) & 0x001F)
```

Sie sehen so sehr schön, wie Funktionsmakros definiert werden. Im Prinzip ist das reine Textverarbeitung. Wenn Sie im Programm zum Beispiel MENUNUM(3) schreiben, macht der Präprozessor Ihres Compilers daraus *3 & 0x1F*. Er setzt also das Argument ein, er kann aber auch – wie die nächsten beiden Makros zeigen – damit rechnen. Im Gegensatz zu echten Funktionen, die nur einmal im Programm stehen, werden also Funktionsmakros immer wieder neu in das Programm eingefügt. Das kostet natürlich Speicher. Damit Sie das erkennen können – hier kommen die Makros ja aus einem Include-File –, werden sie meistens (leider nicht immer) groß geschrieben.

Damit sind dann so einfache Abfragen wie

```
if ( MENUNUM(code) = 3 ) /* in Menü 3? */
```

möglich. Wir machen das eleganter mit *switch* und *case*. Beachten Sie, daß nach den inneren »cases« auch immer ein *break* steht. Das ist nötig, weil ein *break* immer nur aus einem *switch* herausführt, wir aber nach einer Aktion ganz aus allen Abfragen heraus müssen. Beachten Sie ferner, daß vor dem Abruch noch ein

```
ClearMenuStrip(Window);
```

steht, womit die Menü-Leiste wieder entfernt wird. Sie können aber auch mitten im Programm damit das Menü löschen, und es oder ein anderes mit einem erneuten Aufruf von *SetMenuStrip()* wieder einbauen. Doch

damit sind wir schon im Hauptprogramm. Nehmen Sie Listing 2.5.2 aus der fünften Sitzung, und bauen Sie folgende Ergänzungen ein:

Direkt vor die Zeile *void _main()* setzen Sie

```
#include "menu.h"
```

Nach der Kontrolle des Window-Öffnens mit »if (Window == NULL) exit(FALSE);« fügen Sie ein:

```
MakeTheMenu();
SetMenuStrip(Window,&menu[0]);
```

In den Switch-Selektor fügen Sie direkt nach dem CLOSEWINDOW einen neuen *case* ein, was dann so aussieht:

```
case CLOSEWINDOW: close_all();
                  break;
case MENUPICK   : do_menu();
                  break;
```

Damit Intuition das MENUPICK-Event auch meldet, tragen Sie in die open_window-Funktion noch dieses Flag ein, zum Beispiel hinter CLOSEWINDOW | als MENUPICK |.

Workshop

Checkliste

1. Welche Formen von Items gibt es, und welche dürfen keinesfalls einen Command-Key haben?
2. Warum sieht ein Programm, das Makros anwendet, viel kürzer aus, ohne es wirklich zu sein?
3. Wozu braucht man die Funktion *ClearMenuStrip()*?

Ideen für eigene Übungen

1. Erweitern Sie *menu.h* so, daß ein neues Menü »Figuren« mit den Items »Rechteck« und »Kreis« entsteht.

   Kompilieren, linken und testen Sie das Programm.
2. Geben Sie einigen weiteren Menü-Items Command-Keys.

   Kompilieren, linken und testen Sie das Programm.

2.7 Siebte Sitzung: Gadgets für die Mausklicks

Die Themen dieser Sitzung:

- Gadget-Typen
- Strukturen und Flags
- Programmierung von Gadgets

Information

Jede mausorientierte Eingabe mit Ausnahme der Menüs läuft über Gadgets. Wörtlich übersetzt sind das Dingsda, praktisch können es recht komplizierte Gebilde sein. Ein Gadget ist zuerst ein unsichtbares Rechteck. Das Drag-Gadget über der Titelleiste ist ein Beispiel dafür. Normalerweise wird man aber ein Gadget sichtbar machen, indem man ihm einen Rahmen, einen Text (oder beides) oder ein Image (Grafik) gibt. Die Lage des Textes wird relativ zum Gadget angegeben, der Text kann somit auch außerhalb der Gadgets liegen. Der Rahmen selbst ist auch relativ zum (unsichtbaren) Gadget zu sehen, weshalb zum Beispiel seine linke obere Ecke mindestens mit 0,0, besser mit −1,−1 definiert werden sollte. Anwender-Gadgets gehören entweder zu einem Window oder zu einem Requester. Die Lage eines Gadgets wird relativ zu diesem Container angegeben.

Die drei Gadget-Typen

Es gibt drei Gadget-Typen. Das Boolean-Gadget wird typisch für Ja-/Nein-Aussagen verwendet. Kästchen mit Texten wie »OK« oder »Abbrechen« sind Beispiele. Diese Gadgets treten zwar oft als Repräsentanten für TRUE oder FALSE auf, daher der Name, Sie sind aber nicht auf zwei Boolean-Gadgets beschränkt. Ein Taschenrechner zum Beispiel kann aus 20 Boolean-Gadgets bestehen. Das String-Gadget dient zur Eingabe von Texten oder Zahlen. Das Proportional-Gadget stellt Schieberegler dar. Beispiele sind die Farbregler in Preferences.

```
struct Gadget
{
 struct Gadget *NextGadget; /* --> Nachfolger */
 SHORT LeftEdge, TopEdge;/* linke obere Ecke */
 SHORT Width, Height;     /* Breite und Höhe */
 USHORT Flags;            /* siehe unten */
 USHORT Activation;       /* Auch Flags, s. unten */
 USHORT GadgetType;       /* der Typ */
 APTR GadgetRender;       /* --> Border oder
                             Image oder NULL */
 APTR SelectRender;       /* --> oder Image, das
                             nach Anwahl */
 struct IntuiText *GadgetText; /* Text des Gadgets
*/
 LONG MutualExclude;          /* z.Z. unbenutzt */
 APTR SpecialInfo;  /* --> Zusatz-Struktur
                       bei String- und Prop-Gadgets */

 USHORT GadgetID;       /* Kenn-Nummer */
 APTR UserData;         /* Anwender-Erweiterung */
};
```

Listing 2.7.1 Die Gadget-Struktur

Datenstrukturen und Flags

Ein Gadget ist eine Struktur (was hätten Sie erwartet?), die Listing 2.7.1 zeigt. Wie Sie sehen, enthält diese Struktur wieder diverse Zeiger auf andere Strukturen. Für ein unsichtbares Gadget müssen Sie all diese Zeiger auf NULL setzen. Für ein Boolean-Gadget brauchen Sie zumindest in **GadgetText* einen Zeiger auf eine Textstruktur. Praktisch werden Sie einen Rahmen um diesen Text zeichnen (genau: um das Gadget), wofür Sie in *GadgetRender* den Zeiger auf eine Border-Struktur eintragen. Bei einem String-Gadget müssen Sie in *SpecialInfo* die Adresse einer StringInfo-Struktur eintragen und für ein Proportional-Gadget die von *PropInfo*. Die Darstellung des Gadgets erfolgt immer gemäß dem Bild, auf das *GadgetRender* zeigt. Das Objekt kann ein *Image* (Grafik) oder ein *Border* (Polygon, Vieleck) sein.

Flags und Activation-Flags

Es gibt Unmengen von Flags, hier benutzen wir nur GADGHCOMP, womit das Gadget bei der Anwahl komplementiert wird. Die Activation-Flags legen fest, welche Events Intuition meldet, wie sich Gadgets verhalten sollen und was sie bewirken. Wir brauchen hier nur wenige. RELVERIFY bewirkt, daß Events nur gemeldet werden, wenn der Anwender die Maustaste über dem Gadget losläßt. GADGIMMEDIATE heißt sofortige Meldung. STRINGCENTER heißt: Der Text in einem String-Gadget wird zentriert. Für die Proportional-Gadgets brauchen wir noch AUTOKNOB (automatischer Schieber) und FREEHORIZ (horizontaler Schieber).

Die zusätzlichen Datenstrukturen

Bei String-Gadgets muß »SpecialInfo« auf eine Struktur namens *StringInfo* zeigen. Wie diese aufgebaut ist, zeigt Listing 2.7.2.

```
 struct StringInfo
{
 UBYTE *Buffer; /* Arbeitspuffer */
 UBYTE *UndoBuffer; /* Kopie Puffer vor Editierung
*/
 SHORT BufferPos; /* Zeichen-Position im Puffer */
 SHORT MaxChars; /* Puffer-Größe in Zeichen */
 SHORT DispPos; /* Position des 1. sichtb. Zeichens */

 /* Die folgenden Felder werden von Intuition
     aktualisiert: */
 SHORT UndoPos; /* Zeichen-Position im UndoBuffer */
 SHORT NumChars; /* Anzahl Zeichen im Buffer */
 SHORT DispCount; /* Anzahl Zeichen im Gadget */
 SHORT CLeft, CTop; /* linke obere Ecke des Gadgets */
 struct Layer *LayerPtr; /* RastPort des Gadgets */
 LONG LongInt; /* Ergebnis beim Integer-Gadget hier */
 struct KeyMap *AltKeyMap; /* --> Key-Map oder NULL */
};
```

Listing 2.7.2 Die StringInfo-Struktur

Wie Sie nachher im Beispiel-Programm sehen werden, reicht es, wenn Sie die ersten vier Felder initialisieren. Mit Listing 2.7.3 kommen wir zur *PropInfo*, also der Struktur, auf die *SpecialInfo* im Falle von Proportional-Gadgets zeigen muß. Keine Angst vor viel Arbeit, es reicht praktisch, wenn Sie nur drei Felder initialisieren.

```
struct PropInfo
{
 USHORT Flags; /* z.B. AUTOKNOP | FREEHORIZ */
 USHORT HorizPot; /* Stellung des horiz. Potis
                     in Prozent */
 USHORT VertPot; /* bzw. vertikal */

 USHORT HorizBody;/* Schrittweite horizontal */
 USHORT VertBody; /*                 vertikal */

 /* Die folgenden Felder setzt Intuition: */
 USHORT CWidth; /* Container Width */
 USHORT CHeight; /* Container Height */
 USHORT HPotRes, VPotRes; /* Auflösung des Potis */
 USHORT LeftBorder; /* Container- */
 USHORT TopBorder;  /* Position   */
};
```

Listing 2.7.3 Die PropInfo-Struktur

Praxis

Wir wollen unser Programm aus der sechsten Sitzung wie folgt erweitern:

- Nach Anwahl von *Projekt --> Laden* erscheint ein String-Gadget mit der Frage nach dem Dateinamen. »String-Gadget mit Return verlassen« ist ein Event, das angezeigt wird, dito wird dann der Pufferinhalt ausgegeben.
- Nach Anklicken der Close-Box des Windows folgen die Fragen »Wirklich schließen?« und dazu zwei Boolean-Gadgets mit »Nein« und »Jawohl«.
- Im Window sind die drei Proportional-Gadgets »Rot«, »Grün« und »Blau«, mit denen Sie die Schirmfarbe ändern können.

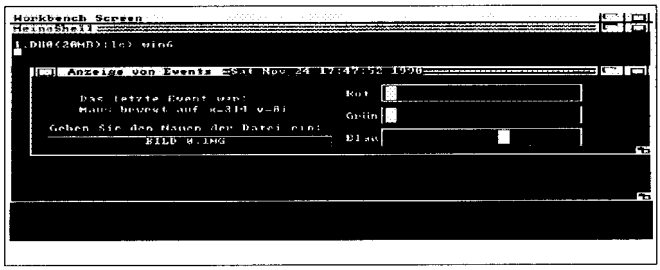


Bild 2.7.1: Das ist unser Ziel

Gadgets programmieren

Wie schon mit den Menüs in der vorigen Sitzung habe ich auch hier alles zum Thema im Include-File *gadget.h* laut Listing 2.7.4 untergebracht.

```
/* gadget.h

   Daten und Funktionen zum
   Anlegen von Gadgets
*/

APTR   iadr;
struct Gadget *gad, gadget[11];
struct IntuiText gtext[11];
struct StringInfo info;
char dobuffer[80], undobuffer[80];

/* für die Rahmen der Gadgets */
SHORT cords1[] = {0,0, 282,0, 282,12, 0,12, 0,0};
struct Border border1 =
                {-2,-2,1,0,JAM1,5,&cords1[0],NULL};
SHORT cords2[] = {0,0, 101,0, 101,21, 0,21, 0,0};
struct Border border2 =
                {-1,-1,1,0,JAM1,5,&cords2[0],NULL};

/* für Prop-Gadget: */
struct Image img[3];
struct PropInfo prop[3];
```

Listing 2.7.4: (Fortsetzung folgende Seiten)

```
/* Farbe in 16 Schritten: */
#define STEP (0xFFFF/0x10)

/* Text in eine IntuiText-struct
   eintragen
*/
void make_gtext(struct IntuiText *name,
           char *text, SHORT left, SHORT top)
{
  name->FrontPen = 1;
  name->BackPen =  0;
  name->DrawMode = JAM1;
  name->LeftEdge = left;
  name->TopEdge = top;
  name->ITextFont = 0;
  name->IText = (UBYTE *) text;
  name->NextText = NULL;
}

/* Gaget-Struktur initialisieren
*/
void make_gadget(char *gtext,
                 struct IntuiText *gname,
                 SHORT tleft, SHORT ttop,
                 struct Gadget *gadget,
                 struct Gadget *next,
                 SHORT left, SHORT top,
                 SHORT width, SHORT height,
                 USHORT flags,
                 USHORT activation,
                 USHORT type,
                 APTR *render,
                 struct IntuiText *text,
                 APTR  *info,
                 USHORT id )
{
```

Listing 2.7.4: (Fortsetzung folgende Seiten)

```
    gadget->NextGadget    = next;
    gadget->LeftEdge      = left;
    gadget->TopEdge       = top;
    gadget->Width         = width;
    gadget->Height        = height;
    gadget->Flags         = flags;
    gadget->Activation    = activation;
    gadget->GadgetType    = type;
    gadget->GadgetRender  = (APTR) render;
    gadget->SelectRender  = NULL;

    make_gtext(gname, gtext, tleft, ttop);
    gadget->GadgetText    = gname;

    gadget->MutualExclude = NULL;
    gadget->SpecialInfo   = (APTR)info;
    gadget->GadgetID      = id;
    gadget->UserData      = NULL;
}

/* Alle Gadgets anlegen
*/
void MakeTheGadgets(void)
{

/* Puffer für String-Gadget initialisieren */
strcpy(dobuffer,"BILD_0.IMG");
info.Buffer = (UBYTE *) dobuffer;
info.UndoBuffer = (UBYTE *) undobuffer;
info.MaxChars = 80;
info.BufferPos = 0;
info.DispPos = 0;

/* Das String-Gadget selbst: */
make_gadget("Geben Sie den Namen der Datei
            ein: ",
            &gtext[0], 0, -15,
```

Listing 2.7.4: (Fortsetzung folgende Seiten)

```
            &gadget[0], NULL ,20,80,280,11,
            GADGHCOMP,
            STRINGCENTER | RELVERIFY,
            STRGADGET,
            (APTR *) &border1, &gtext[0],
            (APTR *) &info, 1 );

/* Das erste Bool-Gadget: */
make_gadget("Nein", &gtext[1], 40,7,
             &gadget[1],&gadget[2],
             180,130,100,20,
             GADGHCOMP,
             GADGIMMEDIATE | RELVERIFY,
             BOOLGADGET,
             (APTR *) &border2, &gtext[1],
             NULL, 2 );

/* Das zweite Bool-Gadget: */
make_gadget("Jawohl", &gtext[2], 25,7,
             &gadget[2],NULL,
             350,130,100,20,
             GADGHCOMP,
             GADGIMMEDIATE | RELVERIFY,
             BOOLGADGET,
             (APTR *) &border2, &gtext[2],
             NULL, 3 );

/* Es folgen drei Prop-Gadgets: */

make_gadget("Rot", &gtext[3], -35,7,
             &gadget[3], &gadget[4],
             350,20,200,20,
             GADGHCOMP,
             GADGIMMEDIATE,
             PROPGADGET,
             (APTR *) &img[0], &gtext[3],
             (APTR *) &prop[0], 4 );
```

Listing 2.7.4: (Fortsetzung folgende Seiten)

```
prop[0].Flags      = AUTOKNOB | FREEHORIZ;
prop[0].HorizBody = STEP;
prop[0].HorizPot   = 0;

make_gadget("Grün",&gtext[4], -35,7,
            &gadget[4], &gadget[5],
            350,45,200,20,
            GADGHCOMP,
            GADGIMMEDIATE,
            PROPGADGET,
            (APTR *) &img[1], &gtext[1],
            (APTR *) &prop[1], 5);
prop[1].Flags      = AUTOKNOB | FREEHORIZ;
prop[1].HorizBody = STEP;
prop[1].HorizPot   = 0;

make_gadget("Blau",&gtext[5],-35,7,
            &gadget[5], NULL,
            350,70,200,20,
            GADGHCOMP,
            GADGIMMEDIATE,
            PROPGADGET,
            (APTR *) &img[2], &gtext[2],
            (APTR *) &prop[2], 6);
prop[2].Flags      = AUTOKNOB | FREEHORIZ;
prop[2].HorizBody = STEP;
prop[2].HorizPot   = 10*STEP;

} /*End MakeTheGadgets() */

/* Gadget-Events auswerten und passende
   Aktionen ausführen
*/
void do_gadget()
{
  switch( ( ( (struct Gadget *)
          iadr) ->GadgetID) )
```

Listing 2.7.4: (Fortsetzung folgende Seite)

```
  {
   case 1: print(20,70,"String-Gadget
                        mit Return verlassen");
           print(20,20, /* 29 Blanks */
           "                                ");
           print(20,20, (char *) &info.Buffer[0]);
           RemoveGadget(Window, &gadget[0]);
           break;

   case 2: RemoveGadget(Window, &gadget[1] );
           RemoveGadget(Window, &gadget[2] );
           SizeWindow(Window,0,-100);
           break;
   case 3: close_all();
            break;
   case 4:
   case 5:
   case 6: SetRGB4(&Screen->ViewPort,0,
                   prop[0].HorizPot/STEP,
                   prop[1].HorizPot/STEP,
                   prop[2].HorizPot/STEP);
            break;
  }
}
```

Listing 2.7.4: Alles für die Gadgets

Die Strukturen *cords* und *border* werden nur benötigt, um die Rahmen der Gadgets zu zeichnen, mehr dazu in der neunten Sitzung. Die Funktion *make_gtext()* unterscheidet sich nur durch die Penfarben von *make_text()* in *menu.h*. Das Problem: Mit den Farben für die Menüs lassen sich keine Gadget-Texte malen, umgekehrt keine Menüs. Da kommt noch eine Aufgabe auf Sie zu.

Die Funktion *make_gadget()* wendet dieselbe Technik an wie *make_item()* aus der vorherigen Sitzung, nur daß hier die Gadget-Strukturen initialisiert werden.

Für das String-Gadget muß einiges vorab getan werden. Zuerst muß das Programm zwei Puffer, nämlich *char dobuffer[80]* und *undobuffer[80]* zur Verfügung stellen. In den *dobuffer* muß der String kopiert werden, der

nach dem Start im Gadget stehen soll. Den *undobuffer* braucht Intuition, weil der Anwender ja mit [Amiga] + [Q] den alten Inhalt wieder herstellen kann. Die Zeiger auf beide Puffer müssen in die info-Struktur eingetragen werden und deren Adresse schließlich in die Gadget-Struktur. Ansonsten unterscheidet sich der Aufwand nicht von den beiden folgenden Boolean-Gadgets, nur daß hier anstatt des Zeigers auf *info* NULL steht. Die beiden Boolean-Gadgets gehören zusammen, weshalb der erste auf das zweite zeigt. Sie können hier wie bei den Menüs beliebig lange Listen bilden, nachher werden alle Gadgets auf einmal gezeichnet.

Die folgenden drei PROPGADGET für die Farbregler bilden auch eine Liste. Hier muß zusätzlich die prop-Struktur mit Daten für den Schieberegler geladen werden. In dieser Struktur ist der Wert von *HorizPot* die aktuelle Stellung des Knopfes. Das ist ein 16-Bit-Wert zwischen 0 und 0xFFFF. Auch der HorizBody-Wert für die Schrittweite ist ein 16-Bit-Wort. Sind beide Werte gleich, hat der Regler nur eine feste Stellung, hat *HorizBody* den halben Wert, sind zwei Stellungen möglich, der Regler liefert dann die Werte 0 und 1. Für unser Beispiel gilt: Wir wollen die Farben einstellen. Die Farbwerte dürfen nur zwischen 0 und 15 liegen. Der Regler darf also nur 16 Werte liefern. In diesem Fall teilt man 0xFFFF/0x10 (65536/16) = 0x1000 (4096). Die Body-Komponente wird also auf 4096 gesetzt. Genau das ist die Konstante STEP, die mit »#define STEP (0xFFFF/0x10)« definiert wurde. Wenn man dann später die Reglerstellung aus der HorizPot-Komponente abliest, muß man deren Wert auch durch STEP teilen und kann damit die Farbe setzen. Die Farbeinstellung erfolgt mittels *SetRGB4()*. Das ist schon wieder ein Vorgriff, den Sie mir hoffentlich verzeihen.

Gadgets abfragen

Wenn jemand ein Gadget anklickt, bekommen wir von Intuition u.a. eine GADGETDOWN-Message und wenn die Maustaste über dem Gadget losgelassen wird GATGETUP. Doch das reicht nicht. Das letzte Argument in den make_gadget()-Aufrufen ist die von uns vergebene Gadget-ID (eine Zahl zwischen 1 und 6). Nur darüber können wir erkennen, welches Gadget angeklickt wurde. Doch diese Zahl meldet uns Intuition leider nicht, weshalb wir einige Umwege gehen müssen. In der Message-Struktur unter *msg->.IAddress* steht die Adresse der Struktur, die ein Event ausgelöst hat. In unserem Fall ist das die Adresse eines Gadgets. Die interessiert uns herzlich wenig, wir wollen die ID wissen. Diese ID (Identifikation) ist die laufende Nummer, die wir selbst im Feld *GadgetID* eingetragen

haben. Darauf müssen wir also zugreifen, und das geschieht in der Funktion *do_gadget()* etwas trickreich. Wir könnten mit

```
gad = (struct Gadget *) iadr;
```

den Zeiger auf die Struktur holen und ihn in *gad* speichern. Nun könnten wir daraus die ID holen, wir können sie aber auch gleich in den *switch* packen, also:

```
switch(gad->GadgetID)
```

Das kann man natürlich trickreicher lösen, was halten Sie davon?

```
int i;
i = (((struct Gadget *)iadr)->GadgetID);
switch(i)
```

Nicht trickreich genug? Nun denn, hier ist die »Endlösung«:

```
switch((((struct Gadget *)iadr)->GadgetID))
```

Gadgets in das Window bringen

Nun zum Hauptprogramm: Gegenüber dem letzten Listing *win3.c* gibt es so viele Erweiterungen, daß ich mit Listing 2.7.5 lieber die Komplettlösung zeige.

```
/* win5.c

   Erweiterung von win3.c um
   Menüs und Gadgets
*/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <time.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;
```

Listing 2.7.5: (Fortsetzung folgende Seiten)

```
struct Screen *Screen;
struct RastPort *rp;

#include "stdwindow.h"

struct IntuiMessage *msg;
ULONG class;
USHORT code;
char string[81];
SHORT mx, my;
struct tm *tp;
time_t  t;
void print(int x, int y, char *string)
{
   SetAPen(rp, 1L);
   Move(rp, x,y);
   Text(rp,
   "                                    ",30L);
   Move(rp, x,y);
   Text(rp, string, strlen( string) );
}

#include "menu.h"
#include "gadget.h"

void _main()
{
  open_libs();
  MakeTheGadgets();
  Window = (struct Window *) open_window
          (
            20,20,619,100,
            " Anzeige von Events ",
            WINDOWCLOSE |
            WINDOWSIZING |
            WINDOWDRAG |
            WINDOWDEPTH |
```

Listing 2.7.5: (Fortsetzung folgende Seiten)

```
            REPORTMOUSE |
            ACTIVATE,
            CLOSEWINDOW |
            GADGETUP | GADGETDOWN |
            NEWSIZE |
            MOUSEMOVE |
            VANILLAKEY |
            INTUITICKS |
            DISKREMOVED |

            MENUPICK,
            &gadget[3] /* Nun nicht NULL */
          );

if (Window == NULL) exit(FALSE);
/* Farben auf "normal": */
Screen = Window->WScreen;
SetRGB4(&Screen->ViewPort,0,0,0,10);

MakeTheMenu();
SetMenuStrip(Window,&menu[0]);

rp = Window->RPort;

print(50,38,"Das letzte Event war:");

for(;;)  /* FOREVER */
 {
   Wait(1L << Window->UserPort->mp_SigBit);

   while( msg = (struct IntuiMessage *)
                GetMsg(Window->UserPort) )
   {
     class = msg->Class;
     code  = msg->Code;
     mx    = msg->MouseX;
     my    = msg->MouseY;
```

Listing 2.7.5: (Fortsetzung folgende Seiten)

```
iadr   = msg->IAddress;
ReplyMsg( msg );

switch( class)
{
 case CLOSEWINDOW:
   if (Window->Height == 100)
      SizeWindow(Window, 0,100);
   AddGadget(Window, &gadget[1], -1);
   AddGadget(Window, &gadget[2], -1);
   RefreshGadgets( &gadget[1], Window, NULL);
    print(240,110, "Wirklich schließen?");
   break;

   case GADGETDOWN :
   case GADGETUP   : do_gadget();
                     break;

  case MENUPICK :
    if(MENUNUM(code) == PROJEKT &&
                     ITEMNUM(code) == LADEN)
 {
   print(20,70, /* 34 Blanks */
      "                                  ");
   AddGadget(Window, &gadget[0], -1);
   RefreshGadgets(&gadget[0], Window, NULL);
     }
    else do_menu();
   break;

  case NEWSIZE :
    print(50,50,"Größe geändert");
    break;
   case MOUSEMOVE  :
   sprintf(string,
       "Maus bewegt auf x=%d y=%d", mx, my);
   print(50,50,string);
```

Listing 2.7.5: (Fortsetzung folgende Seite)

```
            break;
          case VANILLAKEY :
            sprintf(string, "Taste %c gedrückt",code);
            print(50,50,string);
            break;
          case INTUITICKS :
            time(&t);
            Move(rp, 200, 7);
            Text(rp, asctime( localtime(&t)),24 );
            break;
          case DISKREMOVED :
            print(50,50,"Diskette ausgeworfen");

          }
        }
      }
}
```

Listing 2.7.5 Unser Window hat Menüs und Gadgets

In die NewWindow-Struktur muß unter *FirstGadget* die Adresse des ersten Gadgets eingetragen werden, wenn es und seine Nachfolger gleich nach dem Öffnen des Windows auf dem Schirm sein sollen. Das trifft für unsere Farbregler zu. Die anderen Gadgets sollen erst bei Bedarf erscheinen. Das machen die Funktionen *AddGadget()* (Aufnahme in die Gadget-Liste) und *RefreshGadgets()* (Gadgets neu zeichnen). Ist die Aktion gelaufen, muß man mit *RemoveGadget()* die Gadgets wieder deaktivieren.

Im *case* CLOSEWINDOW arbeite ich mit einem Trick. Das Window hat eine Höhe von 100, die Gadgets starten aber bei 130. Deshalb wird zuerst mit *SizeWindow()* das Fenster um 100 nach unten gezogen. Weil das wieder passieren würde, wenn jemand nochmals auf das Close-Gadget klickt, und dann der Amiga abstürzt, wird die Aktion nur zugelassen, wenn das Window noch 100 hoch ist. Wenn der Anwender auf »Nein« klickt, geht das Window wieder auf seine alte Höhe. Beachten Sie, daß in *SizeWindow()* die beiden Zahlen Delta-Werte für Breite und Höhe sind. Bei positiven Zahlen wird das Window größer, bei negativen kleiner.

Der ganze Umstand ist nötig, weil auch deaktivierte Gadgets so lange sichtbar bleiben, bis man das Window neu zeichnet. Mit Gadgets in Requestern – dem Thema der nächsten Sitzung – ist das einfacher.

Workshop

Checkliste

1. Worauf müssen die Zeiger der Gadget-Struktur zeigen, wenn das Gadget unsichtbar sein soll?
2. Welches Gadget braucht eine StringInfo-Struktur?
3. Darf eine Liste von Boolean-Gadgets mehr als zwei Objekte haben?

Ideen für eigene Übungen

1. Erweitern Sie *gadget.h* so, daß neben den Gadgets »Nein« und »Jawohl« ein drittes mit »Vielleicht« erscheint. Wenn dieses Gadget angeklickt wird, soll mit der print()-Funktion eine Meldung wie »Na was denn nun« ausgegeben werden.

   Kompilieren, linken und testen Sie das Programm.
2. Ändern Sie das Programm so, daß die Farbregler-Gadgets erst erscheinen, wenn im Menü »Farben« das Item »Rot« angewählt wird.

   Kompilieren, linken und testen Sie das Programm.

2.8 Achte Sitzung: Requester und Alerts

Die Themen dieser Sitzung:

- Was sind Requester?
- Requester-Arten und Zeichenmodi
- IDCMP-Flags für Requester
- Requester und Gadgets
- Die Requester-Struktur
- Requester-Flags
- Programmieren von Requestern
- Alert-Anwendung

Information

Requester sind im Prinzip nichts weiter als eine Ansammlung von Gadgets in einem gemeinsamen Container, der hier Requester heißt. Es gibt jedoch einige grundsätzliche Unterschiede zu den »freien« Gadgets.

Requester erscheinen nur auf der Bildfläche, wenn das Programm die Funktion *Request()* aufruft. Es gibt noch eine Spezialität, nämlich den Requester, der durch einen Doppelklick auf die rechte Maustaste aktiviert werden kann. Aber auch das geht nur, wenn das Programm diese Aktion zuläßt.

Ist ein Requester sichtbar, wird jeder Input in das Window blockiert (Ausnahme NOISYREQ, siehe unten). Es sind dann nur noch Eingaben in den Requester möglich. Dieser Zustand besteht so lange, bis eines der Gadgets angeklickt wird, das als »Ende-Gadget« markiert ist. Doch da ein Requester immer an ein Window gebunden ist, können Sie natürlich trotz des aktiven Requesters ein anderes Window anklicken. Wenn Sie zum Beispiel ein Requester nach einem File-Namen fragt, und Sie wissen den nicht, können Sie in das CLI-Window gehen und dort den DIR-Befehl geben.

Im Gegensatz zum Input ist die Ausgabe eines Fensters nicht durch einen Requester blockiert. Das kann unter Umständen Probleme ergeben, weil

dann mit einer gewissen Wahrscheinlichkeit dieser Output einen Requester überschreiben kann. Natürlich stellt Intuition auch hierfür Mittel bereit, um solche Pannen abzublocken.

Neben den Requestern Ihres Programms gibt es noch sogenannte System-Requester. Wenn Sie zum Beispiel auf eine Diskette kopieren wollen, die sich in keinem Laufwerk befindet, werden Sie per System-Request aufgefordert, diese Diskette einzulegen. Sie können aber auch selbst – und sehr einfach – System-Requester programmieren. Es hindert Sie auch niemand daran, als Aktion auf eine Requester-Antwort ein weiteres Requester aufzubauen. Diesem kann dann noch einer folgen und noch einer usw. Da aber immer der letzte Requester aktiv ist, müssen Sie diese Kette rückwärts abbauen.

Requester-Arten und Zeichen-Modi

Prinzipiell gibt es zwei Arten von Requestern, nämlich Auto-Requester und Anwender-Requester. Einen Auto-Requester können Sie mit einem einzigen Aufruf der Funktion *AutoRequest()* erscheinen lassen. Wenn Sie zum Beispiel Disketten auf der Workbench duplizieren, sind alle Aufforderungen wie »<From>-Disk einlegen« Auto-Requester. Bei Anwender-Requestern müssen Sie alles selber machen, zum Beispiel, wie in der vorigen Sitzung gezeigt, die Gadgets aufbauen. Einziger Unterschied: Sie kommen nicht in ein Window, sondern in einen Requester-Rahmen.

Eine Abart des Anwender-Requesters ist das Double-Menu-Requester«. Prinzipiell sieht das genauso aus. Der Unterschied ist lediglich, daß dieses Requester aktiviert wird, wenn Sie die Menü-Taste (die rechte Maustaste) doppelklicken. Dazu muß das Requester nicht mit *Request()*, sondern mit *SetDMRequest()* aufgerufen (hier scharfgemacht) werden. Ab diesem Zeitpunkt wartet Intuition auf seine Chance, das Requester zu zeigen. Mit *ClearDMRequest()* wird der Zustand wieder aufgehoben.

Das Zeichnen von Requestern kann auch auf zwei Arten erledigt werden, nämlich Intuition zeichnet oder das Programm zeichnet. Wenn Intuition uns die Arbeit abnehmen soll, müssen wir nur wenige Angaben in die Requester-Struktur eintragen, primär die Adresse der Gadget-Liste und der Text-Struktur.

Die Alternative ist ein Anwender-BitMap-Requester. Dazu stellen Sie eine (natürlich tolle) Grafik zur Verfügung. Eine Gadget-Liste braucht dieses Requester auch, doch die Gadgets müssen unsichtbar sein (Text- und

Image-Zeiger = NULL). Die Gadgets müssen (absolute Pflicht) über Symbolen liegen, die Sie in der BitMap vorgesehen haben.

Lage der Requester

Ein Requester liegt mit seiner linken oberen Ecke immer realativ zur linken oberen Ecke des Windows. Ist das Window kleiner als der Requester, wird nur ein Teil des Requester abgebildet. Da dies katastrophale Folgen haben kann – im Extremfall sieht man den Requester nicht, aber das Programm hängt –, sollten Sie diesen Fall abfangen (IDCMP-Flag (siehe unten) oder eigenes Window).

Relative Requester werden von der derzeitigen Intuition-Version nur bei Double-Menu-Requestern unterstützt. Die Wirkung ist, daß der Requester mit dem Offset in *RelLeft* und *RelTop* relativ zur aktuellen Position des Mauszeigers erscheint, wenn das Flag POINTREL gesetzt ist. Steht der Mauszeiger ungünstig, verschiebt Intuition den Requester auf jeden Fall so weit, daß er noch im Window erscheint.

IDCMP-Flags für Requester

Das wichtigste Flag ist REQVERIFY. Dieses Flag schützt Sie vor einem überraschend auftauchenden Requester. Das kann ein System-Requester oder ein Double-Menu-Requester sein. Ist das Flag gesetzt, erhalten Sie nur eine Message der Klasse REQVERIFY, wenn ein Requester verlangt wird. Gezeichnet wird es erst, wenn Sie die Message mit *ReplyMsg()* beantworten.

Mit REQSET erhalten Sie eine Nachricht, wenn das erste Requester in einem Window gezeichnet wurde. Mittels REQCLEAR können Sie erfahren, wann das letzte Requester verschwunden ist.

Requester-Gadgets

Zuerst folgen diese Gadgets den Regeln für Gadgets aus der Sitzung 7. Drei Unterschiede gibt es allerdings.

- Jedes Gadget muß vom Typ REQGADGET sein.
- Der Zeiger auf die Gadget-Liste wird in der Requester-Struktur notiert.
- Jedes Gadget, das einen Ausstieg aus einem Requester ermöglicht, muß im Activation-Feld das Flag ENDGADGET gesetzt haben. Mindestens ein Gadget muß ein ENDGADGET sein, andernfalls werden Sie das Requester nie los!

Praxis

Natürlich haben wir wieder ein *struct* vor uns, Listing 2.8.1 zeigt die Requester-Struktur. Diese ist wie üblich mit Daten zu füttern. Ist das geschehen, ruft man eine Funktion mit einem Zeiger auf das Requester-struct auf, und das war's dann schon. Doch schauen wir uns zuerst Listing 2.8.1 an.

```
struct Requester
{
 struct Requester *OlderRequest;   /* Vorgänger   */
 SHORT LeftEdge, TopEdge;  /* linke obere Ecke    */
 SHORT Width, Height;      /* Breite und Höhe     */
 SHORT RelLeft, RelTop;    /* Wenn relativ        */
 struct Gadget *ReqGadget; /* Zeiger auf Gadgets  */
 struct Border *ReqBorder; /* Zeiger auf Border   */
 struct IntuiText *ReqText;/* Zeiger auf IntuiText*/
 USHORT Flags;             /* siehe unten         */
 UBYTE BackFill;           /* Füll-Farbe          */
 struct Layer *ReqLayer;   /* Zeiger auf Layer    */
 UBYTE ReqPad1[32];        /* Intuition intern    */
 struct BitMap *ImageBMap; /* Wenn eigenes Image  */
 struct Window *RWindow;   /* Intuition intern    */
 UBYTE ReqPad2[36];        /* Intuition intern    */
};
```

Listing 2.8.1: Die Requester-Struktur

Das meiste davon dürfte klar sein. Für *BackFill* müssen Sie eine Pen-Nummer (Zeichenfarbe) angeben, mit deren Farbe der Requester-Hintergrund gemalt wird. Wählt man diese Farbe geschickt, kann man sich das umhüllende Rechteck (Border) sparen. Auf einem Workbench-Screen ergibt *Backfill = 2* schwarz, *3* orange.

Alle Parameter nach *BackFill* können Sie ignorieren. Auch Flags können Sie auf NULL setzen, und der Requester funktioniert sehr schön. Variieren können Sie mit diesen Flags:

POINTREL	Bei Double-Menu-Requestern erscheint der Requester relativ zum Mauszeiger.
PREDRAWN	Müssen Sie setzen, wenn ReqBMap auf ein eigenes Image zeigt.

NOISYREQ Ermöglicht, daß der Input nicht ganz abgeblockt wird. Tastatur- und Maus-Events werden noch durchgelassen.

Wenn es Sie interessiert, können Sie die folgenden Flags, die Intuition setzt, abfragen:

REQOFFWINDOW Mindestens ein Gadget ist teilweise oder ganz außerhalb des Windows.
REQACTIVE Requester ist aktiv.
SYSREQUEST Ein System-Requester ist aktiv.

Programmieren von Requestern

Am einfachsten kommt man zu einem Requester über die Funktion *AutoRequest()*. Dafür braucht man nicht einmal die Requester-Struktur. Nehmen Sie das Listing 2.7.5 aus der vorigen Sitzung, und bringen Sie zu den globalen Variablen (ziemlich am Beginn des Listings) folgende hinzu:

```
BOOL antwort;
struct IntuiText BodyText, PosText, NegText;
```

Im Hauptprogramm nach der Zeile *SetMenuStrip(...)* fügen Sie ein:

```
make_text( &BodyText, "Wollen Sie wirklich
                       schließen?", 50,20);
make_text( &PosText, "  Ja  ", 6, 3 );
make_text( &NegText, " Nein ", 6, 3 );
```

Nun ändern Sie noch das *case* CLOSEWINDOW, so daß es wie folgt aussieht:

```
antwort = AutoRequest(NULL, &BodyText, &PosText,
                      &NegText, NULL, NULL,
                      350, 100 );
if (antwort)
  close_all();
break;
```

Die erste NULL steht für den Zeiger auf ein Window. Ist dieser Zeiger NULL, macht Intuition ein eigenes Window auf, das den Titel »System Request« hat. Nun folgen die Zeiger auf drei IntuiText-Strukturen. Dabei ist *BodyText* die Frage (Wollen Sie wirklich...), *PosText* die Ja-Antwort (hier Ja) und *NegText* die negative Antwort (hier Nein). Die Funktion ergibt nur dann TRUE, wenn *PosText* angeklickt wurde. Demnach reicht

das *if (antwort)*. Sie können die Variable *antwort* auch sparen, indem Sie gleich *if(AutoRequest(...))* schreiben.

Die beiden nächsten Nullen stehen für *PosFlag* und *NegFlag*. Hier können Sie IDCMP-Flags eintragen, die die Fragen anstatt eines Mausklicks auf das Gadget beantworten. Setzen Sie zum Beispiel für die erste NULL das FLag DISKREMOVED ein, wird das Programm auch beendet, wenn Sie die Diskette auswerfen. »350,100« schließlich sind die Breite und die Höhe des Requesters. Die Maße müssen natürlich zur Länge und Position der Texte passen. Übrigens sind die Argumente »6,3« in *make_text()* die Abstände zu den Rändern der Gadgets, die automatisch gezeichnet werden.

Wenn Sie der Fenstertitel *System Request* stört, können Sie im einfachsten Fall für die erste NULL das vorhandene Window, hier also *Window* einsetzen. Sie können aber auch dafür schnell ein Fenster öffnen, und das geht so: Ändern Sie die Zeile *struct Window *Window;* in *struct Window *Window, *rw;*, womit Sie noch einen Window-Zeiger hätten. Nun schreiben Sie direkt nach dem *case* CLOSEWINDOW: diese Zeilen:

```
rw = (struct Window *) open_window(
     0, 0, 350, 100, "Nachfrage",
     NULL, NULL, NULL);
if (rw == 0) exit(FALSE);
```

Nun müssen Sie dieses Window *rw* natürlich auch schließen, weshalb die Zeilen nach dem *AutoRequest()* dann so aussehen müssen:

```
if (antwort)
  {
    CloseWindow( rw );
    close_all}
else
  CloseWindow(rw);
break;
```

☞ Noch ein Tip: Wenn nur ein Antwort-Gadget gebraucht wird, kann *PosText* NULL sein. Typisch ist das für Meldungs-Requester der Art »Das geht nicht... OK«.

Sie haben richtig erkannt, daß der Auto-Requester höchstens zwei Gadgets haben kann, und das ist oft zu wenig. Sozusagen als erste Übung wollen wir deshalb fast genau denselben Requester noch einmal aufbauen, diesmal allerdings mit eigenen Gadgets. Das sind dann zwar auch nur zwei, doch eine Gadget-Liste kann man beliebig verlängern.

Genau das demonstriert der zweite Requester. Listing 2.8.2 zeigt die Lösung in der Form unseres nächsten Header-Files namens *requester.h*.

```
/* requester.h  */
SHORT cords3[] = {0,0, 69,0, 69,13, 0,13, 0,0};
struct Border border3 = {-1,-1,1,0,
                         JAM1,5,&cords3[0],NULL};

struct Requester request1, request2;
struct IntuiText rtext;

void MakeTheRequester(void)
{
/* Der Requester für "Schließen": */

make_gtext(&rtext, "Wollen Sie wirklich schließen?",
                    50,7);
make_gadget( "Nein ", &gtext[1],40,7,
            &gadget[1],&gadget[2],50,40,100,20,
            GADGHCOMP, GADGIMMEDIATE | ENDGADGET,
            BOOLGADGET | REQGADGET,
            (APTR *) &border2, &gtext[1], NULL, 2);

make_gadget(" Ja ", &gtext[2], 25,7,
            &gadget[2],NULL,190,40,100,20,
            GADGHCOMP, GADGIMMEDIATE | ENDGADGET,
            BOOLGADGET | REQGADGET,
            (APTR *) &border2, &gtext[2], NULL, 3);

request1.LeftEdge  = 30;
request1.TopEdge   = 20;
request1.Width     = 350;
request1.Height    = 70;
request1.ReqGadget = &gadget[1];
request1.ReqBorder = NULL;
request1.ReqText   = &rtext;
request1.Flags     = NULL;
request1.BackFill  = 3; /* Orange */
```

Listing 2.8.2: (Fortsetzung folgende Seite)

```
request1.ImageBMap = NULL;
/* Der DM-Requester für "Farben" */
make_gadget("schwarz", &gtext[6], 3, 1,
            &gadget[6],&gadget[7],5,3,68,12,
            GADGHCOMP, GADGIMMEDIATE | ENDGADGET |
            RELVERIFY,
            BOOLGADGET | REQGADGET,
            (APTR *) &border3, &gtext[6], NULL, 7);
make_gadget("weiß", &gtext[7], 3, 1,
            &gadget[7],&gadget[8], 5,18,68,12,
            GADGHCOMP, GADGIMMEDIATE | ENDGADGET |
            RELVERIFY,
            BOOLGADGET | REQGADGET,
            (APTR *) &border3, &gtext[7], NULL, 8);
make_gadget("rot", &gtext[8], 3, 1,
            &gadget[8],&gadget[9],5,35,68,12,
            GADGHCOMP, GADGIMMEDIATE | ENDGADGET |
            RELVERIFY,
            BOOLGADGET | REQGADGET,
            (APTR *) &border3, &gtext[8], NULL, 9);
make_gadget("blau", &gtext[9], 3, 1,
            &gadget[9],NULL,5,53,66,12,
            GADGHCOMP, GADGIMMEDIATE  | ENDGADGET |
            RELVERIFY,
            BOOLGADGET | REQGADGET,
            (APTR *) &border3, &gtext[9], NULL, 10);
request2.RelLeft    = 0;
request2.RelTop     = 0;
request2.Width      = 87;
request2.Height     = 75;
request2.ReqGadget  = &gadget[6];
request2.ReqBorder  = NULL;
request2.ReqText    = NULL;
request2.Flags      = POINTREL | NOISYREQ;
request2.BackFill   = 2;
request2.ImageBMap  = NULL;
}
```

Listing 2.8.2: Das Header-File requester.h

Das meiste aus dem Listing kennen Sie schon aus der Sitzung 7 (Gadgets). Sie sollten deshalb auch *gadget.h* kopieren und anpassen. Die Gadgets und die Texte werden auch genauso aufgebaut, beachten Sie jedoch die neuen Flags. Sie müssen auf jeden Fall REQGADGET setzen, damit Intuition weiß, daß Sie ein Requester-Gadget wollen. ENDGADGET zu setzen, ist recht praktisch, weil dann der Requester automatisch abgebaut wird, wenn das Gadget angeklickt wird. Sonst müßten Sie das mit *EndRequest(&request, Window);* selbst tun.

☞ Noch ein Trick: Sie sehen, daß ich mit *request.ReqBorder = NULL;* dem Requester keinen Rahmen (Border) spendiert habe. Dennoch fällt er auf, denn mit *request. BackFill = 3;* wird die Hintergrundfarbe auf Orange gesetzt.

Requester einsetzen

Nachdem die Requester-Struktur mit den Daten gefüllt ist, reicht der schlichte Aufruf von

```
Request(&request1, Window);
```

und der Requester ist auf dem Schirm. Das wäre aber in unserem Fall falsch, denn *request1* soll nur erscheinen, wenn das Close-Gadget des Windows angeklickt wird. Doch der Reihe nach. Nachdem Sie Listing 2.8.2 als *requester.h* gespeichert haben, nehmen Sie sich wieder Listing 2.7.5 vor. Dort fügen Sie nach der Zeile *#include "gadget.h"* ein:

```
#include "requester.h"
```

Nach der Zeile *SetMenuStrip(...);* fügen Sie ein:

```
MakeTheRequester();
SetDMRequest( Window, &request2 );
```

Das case CLOSEWINDOW müssen wir nun schon wieder ändern, und zwar so:

```
case CLOSEWINDOW: Request(&request1, Window);
                  do_gadget();
                  break;
```

Die Gadgets werden genauso ausgewertet wie bisher, nämlich über ihre ID. Doch jetzt können wir den ganzen Aufwand streichen, weil Requester mitsamt ihren Gadgets automatisch vom Schirm verschwinden. Deshalb ändern Sie in *gadget.h* (Listing 2.7.4) das *case 2* schlicht in

```
case 2:
        break;
```

Soll heißen: Wenn der User »Nein« anklickt, tun wir gar nichts. Der Requester wird trotzdem automatisch abgebaut, weil wir auch für das Nein-Gadget das Flag ENDGADGET gesetzt hatten.

Wir bauen einen Double-Menu-Requester

Nun zu unserem zweiten Requester, der *request2* heißt. Dieser wurde mit *SetDMRequest (Window, &request2);* aktiviert, aber nichts passiert, oder? Nun, im Namen der Funktion steckt das Kürzel für »Double Menu«, soll heißen, sie müssen die Menü-Taste (die rechte Maustaste) doppelt klicken. In diesem Fall erscheint der Requester auf dem Schirm, und zwar da, wo gerade der Mauszeiger ist. Das hat allerdings nichts mit dem »Double Menu« zu tun, sondern liegt daran, daß ich das Flag POINTREL (zeigerrelativ) gesetzt habe. Das nächste Flag, nämlich NOISYREQ, soll bei der Gelegenheit nur demonstriert werden. Ein »noisy requester« blockt nicht den Input in das Window ab. Wenn Sie zum Beispiel Tasten betätigen, werden die noch angezeigt.

Der Requester selbst bringt untereinander vier kleine Gadgets mit den Texten schwarz, weiß, rot und blau. Das Anklicken eines Gadgets läßt momentan nur den Requester verschwinden. Später werden wir ihn noch nutzen, um ganz schnell die Zeichenfarbe wechseln zu können. Beachten Sie noch einen Trick: Dieser Requester hat keinen »BodyText«. Wenn man diese Überschrift nicht braucht, setzt man einfach *ReqText = NULL*.

Wir geben Alarm

Zum Schluß die simpelste Form von Requestern, der sogenannte Alert (Alarm), wie Sie ihn von den Guru-Meldungen (hoffentlich selten) her kennen. Es gibt zwei Arten von Alerts, nämlich den RECOVERY_ALERT und den DEADEND_ALERT. Letzterer heißt nicht »totes Ende« sondern Sackgasse, also es geht nicht weiter, der Anwender muß neu booten. Ein RECOVERY_ALERT hingegen schließt das Booten aus, Ihr Programm sollte nach diesem Fall ordnungsgemäß enden. Mit Alerts sollten sie nur so kritische Dinge wie »Library oder Window läßt sich nicht öffnen« mitteilen. Ansonsten sind Requester vorzuziehen, die erschrecken die Leute nicht so.

```
/* alert.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;
#include "stdwindow.h"

void _main()
{
  open_libs();

  Window = (struct Window *)
           open_window(0,0,640,256,NULL,
                       BORDERLESS, NULL, NULL);
  if (Window == NULL) exit(FALSE);

  DisplayAlert(RECOVERY_ALERT,
  "\1\0\30 Panik! \0c\1\0\50 Drücke eine Maustaste
   \0\0",70);

  close_all();

}
```

Listing 2.8.3: Alarm mit Alert

Diesmal ist das Listing 2.8.3 kein Header-File, sondern ein solo laufendes Programm. Die Funktion, *DisplayAlert()* hat drei Parameter, nämlich

- den Typ,
- die Meldung und
- die Höhe der Alert-Box

Die Meldung ist ein String, der es in sich hat. Dabei müssen Sie folgende Form beachten:

- X-Position des Textes in zwei Bytes
- Y-Position des Textes in einem Byte
- Der erste Text
- Ein 0-Byte
- Wenn Ende: noch ein Null-Byte
- Wenn nicht Ende: ein Byte ungleich 0
- Nächster Text

Analysieren wir einmal den String

```
"\1\0\30 Panik! \0c\1\0\50 Drücke eine Maustaste
 \0\0"
```

so heißt das:

Teilstring	Bedeutung
"\1\0\	X = 1 * 256 + 0 = 256
30	Y = 30
Panik! \0	Text 1
c	Fortsetzung folgt
\1\0	X = 256
\50	Y = 50
Drücke eine Maustaste\0	Text 2
\0"	Ende

Workshop

Checkliste

1. Muß man die Fragen in einem Requester gleich beantworten, oder darf man vorher noch etwas anderes tun?
2. Mit welchem IDCMP-Flag verhindern Sie, daß ein Requester überschrieben werden kann?
3. Wie verhindern Sie, daß ein Window so klein wird, daß der Requester abgeschnitten wird oder gar ganz verschwindet?

Ideen für eigene Übungen/Erweiterungen

1. Der Aufruf der Funktion

```
AutoRequest(NULL, &BodyText, &PosText,
            &NegText, NULL, NULL,
            350, 100 );
```

   ist etwas umständlich, weil man vorher mit drei Aufrufen von *make_text()* die IntuiText-Strukturen bereitstellen muß. Entwickeln Sie deshalb eine Funktion, die einen Aufruf dieser Art ermöglicht:

```
AutoRe( "Body-Text, "Pos-Text", "Neg-Text" );
```

   Kompilieren, linken und testen Sie das Programm.

2. Entwerfen Sie analog zu *request2* einen Requester für die Strichstärken mit Gadgets, welche die Zahlen 1 bis 4 als Text haben.

2.9 Neunte Sitzung: Und nun wird gemalt

Die Themen dieser Sitzung

- Das Malprogramm
- Makros
- Grafik-Funktionen
- Tricks und Techniken

Information

In dieser Sitzung wird unser Malprogramm fast komplett, nur das Laden und das Sichern der Bilder kommt erst in der zehnten Sitzung an die Reihe. Zuerst müssen Sie natürlich die Aufgabe kennen. Unser Programm hat die folgenden Menüs:

Projekt	Farben	Figuren	Spezial
Neu	Schwarz	Rechteck	Figur füllen
Laden	Weiß	Gefülltes Rechteck	Alles übermalen
Sichern	Rot	Ellipse	Farben regeln
Ende	Blau	Gefüllte Ellipse	

Tabelle 2.9.1: Menüstruktur des Malprogramms

»Ende« und »Alles übermalen« haben noch Sub-Menüs der Art »Wirklich?« und »Lieber doch nicht«. Ferner gibt es die vier Requester laut Bild 2.9.1 mit folgenden Funktionen:

Laden/Sichern:	Ein String-Gadget für den Datei-Namen und zwei BOOL-Gadgets für »Abbruch« und »OK«.
Schließen:	Der Text »Wollen Sie wirklich schließen?« und zwei BOOL-Gadgets für »Nein« und »Ja«.
Farbregler:	Drei Prop-Gadgets für Rot, Grün und Blau sowie zwei BOOL-Gadgets für »Reset« und »OK«.
Farben:	Vier kleine BOOL-Gadgets direkt untereinander mit den vier Farbnamen zur Auswahl. Dieser Requester erscheint an der Position des Mauszeigers, wenn Sie die rechte Maustaste zweimal klicken.

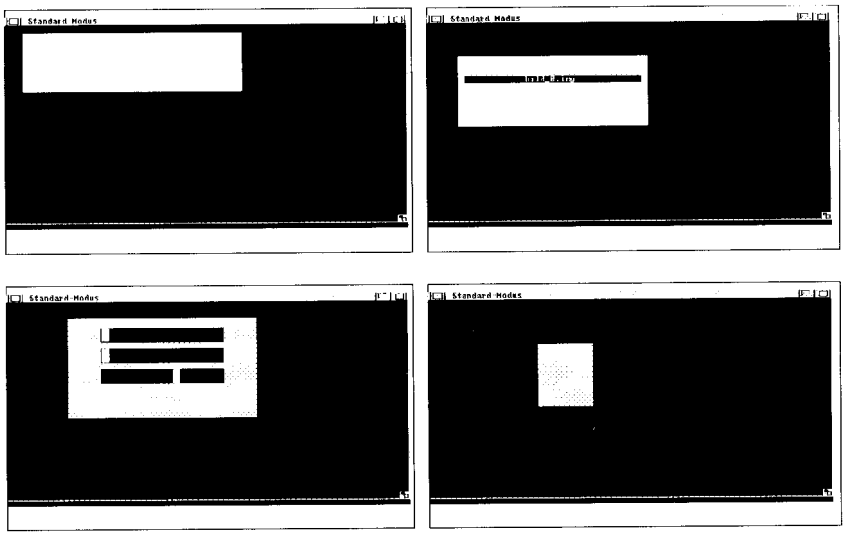


Bild 2.9.1: Die vier Requester des Malprogramms

Der letzte Punkt ist zwar nur das Doppel des Menüs »Farben«, doch das sollen Sie später noch ändern. Solange nichts im Figuren-Menü angewählt ist, können Sie einfach losmalen. Sobald Sie die linke Maustaste drücken und die Maus bewegen, wird in der aktuellen Farbe gezeichnet. Tippen Sie die Taste [w], können Sie nur waagerechte Linien ziehen, nach [s] nur senkrechte. Sie können, die Maus festhaltend, die beiden Tasten wechseln und damit zum Beispiel Treppen zeichnen. Jede andere Taste hebt die Funktionen wieder auf.

Rechtecke und Ellipsen zeichnen Sie so, daß Sie erst die linke obere und dann die rechte untere Ecke anklicken. Die Aufforderungen dazu und überhaupt die Status-Meldungen erscheinen im Fenstertitel. Wie man ein

Rechteck mit der Maus ziehen kann, verrate ich noch, überlasse diese Aufgabe vorläufig Ihnen. Um eine Figur zu füllen, müssen Sie nur das Menü anwählen und dann in die Figur klicken. Gefüllt wird jede beliebige Figur, Sie muß nur geschlossen sein. Wenn nicht, läuft die Farbe über das ganze Window, aber nicht in andere Figuren hinein. »Alles übermalen« hingegen übermalt wirklich das ganze Window mit einer Farbe, die dann sozusagen der Hintergrund ist. Mit dieser Farbe können Sie auch andersfarbige Linien oder Figuren radieren. Der Menüpunkt »Neu« ist eine Variante von »Alles übermalen«. Er zeichnet das Window blau und setzt die Pen-Farbe wieder auf weiß.

Die Menü-Items »Laden« und »Sichern« lassen zwar den Requester erscheinen, doch das »OK« wirkt noch nicht. Das Thema heben wir uns für die nächste Sitzung auf. Ein Trost: Sie müssen dafür nicht mehr ändern, sondern nur noch einiges hinzubringen.

Praxis

Unser Hauptprogramm (Listing 2.7.5 mit den Ergänzungen aus der achten Sitzung) wird nun nochmals erweitert und stellenweise geändert. Damit Sie (und ich) nicht den Überblick verlieren, zeigt Listing 2.9.1 die ganze Lösung.

☞ Für das Programm brauchen Sie die Include-Files (H-Files) *gadget.h*, *requester.h* und *menu.h* auf dem Endstand laut Anhang. Diese Texte wurden geändert. Das gleichfalls benötigte *stdwindow.h* steht dort zu Ihrer Referenz, ist aber nicht neu.

Makros einsetzen

Listing 2.9.1 enthält dieses Makro:

```
#define Titel(t)  SetWindowTitles(Window,(t),-1)
```

Um einen neuen Window-Titel zu setzen, könnte man zum Beispiel schreiben »SetWindowTitles(Window, "Neuer Titel", –1)«. Da das häufig vorkommt, spare ich mit dem Makro einiges an Tipperei. In diesem Fall reicht *Titel("Neuer Titel")*. So ein Funktionsmakro kann beliebig viele Argumente haben, hier ein Beispiel mit zweien:

```
#define printmult(a,b) ( printf("%d", a * b) )
printmult(3,7);
```

Beachten Sie ein paar Regeln: Makros müssen entweder in einer Zeile stehen oder mit dem Zeilenfortsetzungszeichen (\) umbrochen werden. Wird der Makro-Ausruck umbrochen oder enthält er Leerstellen, muß er in Klammern gesetzt werden. Schaden können die Klammern nie. Werden Argumente weiterentwickelt – würde man hier z.B. a * b + a schreiben –, muß man sie in Klammern setzen ((a) * (b) + (a)). Auch hier können ein paar Klammern zuviel nie schaden.

Prototypen deklarieren

Für alle Funktionen des Hauptprogramms wurden Prototypen definiert. Das ist nichts weiter als der Funktionskopf, gefolgt von einem Semikolon. Die Maßnahme hat zwei Vorteile. Zuerst kann der Compiler damit prüfen, ob die Argumente des Funktionsaufrufs vom richtigen Typ sind. Zum zweiten muß man dann keine bestimmte Reihenfolge von Funktionen und H-Files beachten. Daß die H-Files selbst noch in einer bestimmten Folge geladen werden müssen, liegt daran, daß ich da die Prototypen aus Platzgründen gespart habe. Sonst sollte man das nie tun.

Requester richtig anwenden

Im H-File *request.h* hat sich einiges getan. Zuerst wurden alle Gadgets in Requester verlegt. Das hat den Vorteil, daß sie automatisch abgebaut werden und der Hintergrund wieder hergestellt wird. Das String-Gadget für die Filenamen-Eingabe hatte einen Fehler, es fehlten die Gadgets für »Abbruch« und »OK«. Die gibt es nun, dito wurden die Gadgets für die Farbregler um »Reset« und »OK« erweitert. Diese beiden Requester und auch der für »Schließen« werden im Hauptprogramm mit *MakeThe-Requester()* nur aufgebaut, aktiv werden sie erst später.

Die Requester-Strukturen werden jetzt in einem Array vom Typ *struct Requester* gehalten. Neu ist dabei *request[3]).* Das sind zuerst vier kleine Boolean-Gadgets mit den Farbnamen. Dieser Requester erscheint an der Position des Mauszeigers, wenn Sie die rechte Maustaste zweimal klicken. Das tut er, weil er vom Typ POINTREL (zeigerrelativ) ist und weil er mit *SetDMRequest(Window,&request[3]);* initialisiert wurde. In unserer open_window()-Funktion hingegen bleibt der Zeiger auf dem ersten Requester NULL, also erscheint keiner, wenn das Window öffnet. Hier können Sie einen Zeiger auf einen Requester eintragen, der nur zu Programmbeginn erscheint, z.B. »Guten Tag sagen«. Außerdem ist der Zeiger immer für Testzwecke gut. Wenn Sie einen neuen Requester gestalten, tragen Sie ihn hier ein. Erst wenn er gut aussieht, bauen Sie ihn richtig in das Programm ein und setzen den Zeiger wieder auf NULL.

Events holen und auswerten

Unser Window muß bekanntlich auf Events (Ereignisse oder Messages) von Intuition warten und dann darauf reagieren. Am Prinzip hat sich nichts geändert, es bleibt beim *Wait()*, *GetMsg()* usw., doch wurde das jetzt in die Funktion *GetEvents()* verlegt. Sinn der Übung: Wir springen aus der Hauptschleife zu Funktionen, wo wir auch auf Messages warten müssen. In diesem Fall rufen wir dann einfach *GetEvents()* auf.

Auf der Gegenseite steht ein anderes Problem. Nehmen wir als Beispiel die Menüpunkte »Laden« und »Sichern«. In beiden Fällen soll derselbe Requester erscheinen, der nach dem Datei-Namen fragt. Also sagen wir zuerst bei einem Event der Klasse MENUPICK schlicht *do_menu()*. Diese Funktion (in *menu.h*) kommt über die Item-Nummer zu den *cases*:

```
case LADEN   : dflag = 1;
               Request(&request[0], Window);
               break;
case SICHERN: dflag = 2;
               Request(&request[0], Window);
               break;
```

In der Variable *dflag* wird notiert, welches Menü angewählt wurde. Das muß sein, weil mit der nächsten Message das Code-Feld und damit die Item-Nummer einen ganz anderen Wert hat. Nun wird der Requester aufgerufen. Dort kann der User einen Namen eintippen und dann auf ein Gadget klicken. An dieser Stelle warten wir wieder auf eine Message, und zwar jetzt auf eine der Klasse GADGETUP (Maustaste über Gadget losgelassen). In diesem Fall rufen wir *do_gadget()* auf (in *gadget.h*). Dort holen wir die ID des Gadgets und können dann – wenn es die ID von »OK« (Nummer 14) war – mit

```
case 14: do_datei();
         break;
```

schließlich die passende Funktion aufrufen. Sie haben das Problem erkannt? Die Messages treffen nacheinander ein und überschreiben jeweils den Vorgänger. Wir brauchen aber manchmal zwei verschiedene Messages, um einen Fall endgültig bearbeiten zu können.

Mit der Maus zeichnen

So einen Fall hätten wir auch bei der nächsten Aufgabe: Wenn die Maustaste gedrückt UND die Maus bewegt wird, soll gezeichnet werden. Doch leider ist das Ergebnis in der Message-Klasse immer nur »Maustaste betätigt« ODER »Maus bewegt«. Die Lösung sieht im Prinzip so aus:

```
switch( class)
  {
   case MOUSEBUTTONS: notiere ob Taste gedrückt
                      oder losgelassen wurde;
                      break;
   case MOUSEMOVE   : if (Taste gedrückt)
                        zeichne;
                      break;
  }
```

Praktisch wird für den Fall »Taste gedrückt« die Variable *mflag* gleich 1 und sonst gleich 0 gesetzt.

Für das Zeichnen hatte ich anfangs eine tolle Idee, nämlich *WritePixel(rp, mx, my);*. Diese Funktion schreibt einen Bildpunkt, *mx* und *my* sind die Mauskoordinaten, wie sie in *GetEvents()* notiert wurden. *rp* ist ein Zeiger auf dem *RastPort*, den wir uns vorher beschafft hatten. Das funktioniert, hat aber einen Haken. Wenn die Maus schnell bewegt wird, ist die Linie nicht mehr durchgezogen, sondern nur noch gepunktet. Es gibt nur 10 bis 60 Messages pro Sekunde, Sie können aber leicht in einer Sekunde den Mauszeiger über die volle Schirmbreite von 640 Punkten ziehen. Die Lösung heißt *Move()* und *Draw()* und sieht im Prinzip so aus:

```
case MOUSEBUTTONS:
            ...
            x = mx;
            y = my;
            ...
case MOUSEMOVE :
     if (mtaste )
      {
        Move(rp, x,y);
        Draw(rp, mx, my);
        y = my;
        x = mx;
      }
```

Im *case* MOUSEBUTTONS wird in *x* und *y* die Position des Mauszeigers im Augenblick des Drückens der Maustaste notiert. Diese Werte ändern sich so lange nicht, wie die Maustaste niedergehalten wird. Nun kommt der *case* MOUSEMOVE zum Zuge. Mit *Move(rp, x,y);* wird der (unsichtbare) Grafik-Cursor auf diese Position gesetzt. Dann wird mit *Draw(rp, mx, my);* von dort eine Gerade zur aktuellen Mausposition gezogen. Anschließend – und jetzt kommt der Trick – werden *x* und *y* auf diese Mausposition gesetzt. Das folgende *Move()* – wir sind praktisch in einer Schleife – stellt den Grafik-Cusor dahin, und die nächste Gerade wird zur aktuellen Mausposition gezogen. Endergebnis: Wie schnell Sie die Maus auch bewegen, es gibt immer eine Linie, Kurve oder was auch immer, nur daß sie aus einzelnen Geraden zusammengesetzt wird, die im Extremfall auch nur einen Punkt lang sein können (und damit mathematisch keine Geraden mehr sind).

Ein Blick auf das Listing 2.9.1 zeigt, daß da noch mehr in der Schleife passiert. Es gibt da nämlich noch die Klasse VANILLAKEY für die Tastatur, wo ein Druck auf [w] das *wflag=1*, einer auf [s] das *sflag=1* und jede andere Taste beide Variablen auf 0 setzt. Dito schaltet [w] [s] aus und [s] [w] ein. Für das Zeichnen hat das nun eine sinnige Folge. Ist nämlich zum Beispiel *wflag=1*, wird mit

```
if(wflag)
  my = y;
```

einfach die alte y-Position als »Maus-Y« eingetragen. Ergebnis: Die Linie wird immer waagerecht gezogen. Für senkrechte Linien ist sinngemäß das *sflag* zuständig.

Die ROM-Grafik nutzen

Die Zeichen-Funktionen sind sogenannte Grafik-Primitiven, die im Amiga-ROM stehen. *Move()* und *Draw()* kennen Sie schon. Damit zeichnen wir auch Rechtecke aus vier Linien. Gefüllte Rechtecke sind mit *RectFill(rp, x1,y1, x2,y2)* direkter zu erzeugen. Dabei sind *x1,y1* die linke obere und *x2,y2* die rechte untere Ecke. Für Ellipsen gibt es die Funktion *DrawEllipse(rp, x0,y0, xr,yr)*, wobei *x0,y0* den Mittelpunkt und *xr,yr* die beiden Radien angeben. Da wir die Funktion jedoch wie ein Rechteck aufrufen, müssen wir die Werte umrechnen.

Die gefüllte Ellipse zeigt eine andere Technik. *AreaEllipse()* zeichnet nämlich noch gar nichts, sondern nimmt die Figur nur in die Area-Liste auf. Jetzt können noch diverse andere Area-Funktionen folgen. Sie alle

werden erst – und zusammen – gezeichnet, wenn *AreaEnd()* aufgerufen wird. Die Liste ist der Grund, warum wir mit *InitArea()* und den drei folgenden Funktionen die »Area-Info« initialisieren und Speicher dafür beschaffen müssen. Dieser Speicher muß am Programmende wieder freigegeben werden, daher die Zeile *FreeRaster(ram, 640,256)* in der do_close()-Funktion. Bliebe noch das Füllen, was mit *Flood;(rp, 1, mx, my)* geschieht. Damit wird einfach vom Punkt *mx,my* ausgehend die Figur gefüllt. Diese muß geschlossen sein, weil die Funktion alles so lange füllt, wie sie nicht auf schon gesetzte Bildpunkte stößt.

```
    /* malen.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>

#define Titel(t)  SetWindowTitles(Window,(t),-1)

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;
struct Screen *Screen;
struct RastPort *rp;
struct AreaInfo AInfo;     /* Für das Füllen  */
struct TmpRas    TRas;     /* von Bereichen   */
SHORT  Buffer[100];        /* mit einer Farbe */
LONG   ram;
struct IntuiMessage *msg; /* Für das Message- */
ULONG  class;             /* Handling        */
USHORT code;
SHORT  mx, my;    /* Zum Zeichnen */
SHORT x, y;
SHORT mtaste = 0;
SHORT wflag  = 0;
SHORT sflag  = 0;
SHORT dflag  = 0 /* Datei-Modus */
struct Requester request[4] ;
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
/* Prototypen, siehe Listing-Ende
   ------------------------------*/
void GetEvents(void);
void do_rechteck(int fill);
void do_ellipse(int fill);
void do_farben(USHORT code);
void do_fuellen(void);
void do_malen(void);
void do_close(void);
void do_neu(void);
void do_sysreq(char *text);
void do_datei (void);

#include "stdwindow.h"    /* in dieser */
#include "gadget.h"       /* Folge     */
#include "requester.h"    /*           */
#include "menu.h"         /* bitte.    */

void main()
{
open_libs();
Window = (struct Window *) open_window(
         0,0,640,250," Standard-Modus ",
         WINDOWCLOSE |
         WINDOWDEPTH |  /*Window-Flags */
         REPORTMOUSE |
         ACTIVATE,
         CLOSEWINDOW |
         GADGETUP |     /* IDCMP-Flags */
         MOUSEMOVE |
         MOUSEBUTTONS |
         VANILLAKEY |
         MENUPICK,
         NULL); /* Kein Requ. nach Öffnen */
if (Window == NULL) exit(FALSE);
Screen = Window->WScreen; /* Grundfarben */
SetRGB4(&Screen->ViewPort,0,0,0,10);
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
MakeTheMenu(); /* Menü initialisieren */
SetMenuStrip(Window,&menu[0]);

MakeTheRequester(); /* Alle Req. initialisieren */
SetDMRequest(Window,&request[3]);

rp = Window->RPort;
SetAPen(rp, 1L); /* Starte mit weißem Pen */

InitArea( &AInfo, Buffer,10 );
rp->AreaInfo = &AInfo;          /* Init. Area-RAM */
ram = AllocRaster( 640,256 );
rp->TmpRas = ( struct TmpRas *) InitTmpRas(&TRas,
                ram,  RASSIZE( 640,256 ) );

for(;;)    /* Hauptschleife */
 {
  GetEvents();

  switch( class)
    {

     case MOUSEBUTTONS:
     if(code == SELECTDOWN ) /* Wenn linke Maus- */
       {                     /* Taste gedrückt   */
           mtaste = 1;
           x = mx; y = my; /* Mausposition bei  */
       }                   /* beim ersten Klick */
       if (code == SELECTUP )
          mtaste = 0;
     break;
    case MOUSEMOVE : /*Freihandzeichnen */
      if (mtaste )    /* --------------- */
       {
         Move(rp, x,y);
         if(wflag)    /* wenn waagerecht */
            my = y;   /* halte y fest    */
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
        if (sflag)  /* wenn senkrecht */
           mx = x;  /* halte x fest   */
        Draw(rp, mx, my);
        y = my;  /* Endwerte = neuer Startwerte */
         x = mx;
       }
    break;

    case CLOSEWINDOW: /* Frage, ob schließen */
                      Request(&request[1], Window);
                       break;

    case GADGETUP : do_gadget(); /* in "gadget.h" */
                    break;

    case MENUPICK: do_menu(); /* in "menu.h" */
                   break;
    case VANILLAKEY:
                    if (code == 'w' )
                     {
                       wflag = 1;
                       Titel(" Waagerchte Linie ");
                     }
                    else
                      wflag = 0;

                    if (code == 's' )
                     {
                       sflag = 1;
                       Titel(" Senkrechte Linie ");
                     }

                    else
                       sflag = 0;

                    if ( !(wflag | sflag) )
                    Titel(" Standard-Modus ");
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
                    break;
    } /* Ende switch */
  } /* Ende for */
}  /* Ende main */

/* Beginn der Funktionen
   ---------------------*/

void GetEvents(void)
{
  Wait(1L << Window->UserPort->mp_SigBit);
  while( msg = (struct IntuiMessage *)
               GetMsg(Window->UserPort) )
      {
        class = msg->Class;
        code= msg->Code;
        mx= msg->MouseX;
        my= msg->MouseY;
        iadr= msg->IAddress;
        ReplyMsg( msg );
      }
}


void do_rechteck(int fill) /* Rechtecke zeichnen.  */
{                          /* Gefüllt, wenn fill==1*/
  SHORT x1,y1, x2,y2;

  Titel("Klicke linke obere Ecke des Rechtecks");
  code = 0;
  while (code != SELECTDOWN) /* bis Maustaste unten */
    GetEvents();
  x1 = mx; y1 = my;
  WritePixel(rp, x1, y1); /* Startpunkt zeichnen */

  Titel("Klicke rechte untere Ecke des Rechtecks");
  code = 0;
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
  while (code != SELECTDOWN) /* bis Maustaste unten */
    GetEvents();
  x2 = mx; y2 = my;
  if (fill)
    RectFill(rp, x1,y1, x2,y2 );
  else          /* Rechteck mit Linien zeichnen */
   {
     Move(rp, x1,y1);
     Draw(rp, x2, y1 );
     Draw(rp, x2, y2);
     Draw(rp, x1, y2);
     Draw(rp, x1, y1);
   }
  Titel("Standard-Modus");
}


void do_ellipse(int fill) /* Ellipse zeichnen.    */
{                         /* Gefüllt, wenn fill==1 */
  SHORT x1,y1, x2,y2, x0, y0;

  Titel("Klicke linke obere Ecke des
         umgebenden   Rechtecks");
  code = 0;
  while (code != SELECTDOWN)
    GetEvents();
  x1 = mx; y1 = my;
  WritePixel(rp, x1, y1);
  Titel("Klicke rechte untere Ecke des
         umgebenden Rechtecks");
  code = 0;
  while (code != SELECTDOWN)
    GetEvents();
  x2 = mx; y2 = my;
  x0 = x1 + (x2 - x1) / 2;
  y0 = y1 + (y2 - y1) / 2;
  if (fill)
```

Listing 2.9.2: (Fortsetzung folgende Seiten)

```
    {
      AreaEllipse( rp, x0,y0, (x2-x1)/2, (y2-y1)/2);
      AreaEnd(rp);
    }
  else
    DrawEllipse( rp, x0, y0, (x2-x1)/2, (y2-y1)/2);

  Titel("Standard-Modus");
}

void do_farben( USHORT code ) /* Pen-Farbe setzen */
{
  switch(code)
    {
      case SCHWARZ: SetAPen(rp, 2L);
           break;
      case WEISS: SetAPen(rp, 1L);
           break;
      case ROT: SetAPen(rp, 3L);
           break;
      case BLAU : SetAPen(rp, 0L);
           break;
    }
}

void do_fuellen(void)    /* Figur mit Farbe füllen */
{
  Titel("Klicke in die Figur");
  code = 0;
  while (code != SELECTDOWN)
    GetEvents();
  Flood(rp, 1, mx, my);
  Titel("Standard-Modus");
}

void do_malen( void ) /* Das ganze ganze Window */
{                     /* (nicht Menü) übermalen */
```

Listing 2.9.2: (Fortsetzung folgende Seite)

```
 RectFill(rp,Window->LeftEdge+2, Window->TopEdge+11,
          Window->LeftEdge + Window->Width-4,
          Window->TopEdge+Window->Height-2);
}

void do_neu(void) /* Menü-Punkt "Neu" malt das  */
{                 /* Window blau, Pen wird weiß */
  SetAPen(rp, 0L);
  do_malen();
  SetAPen(rp, 1L);
}

void do_close(void) /* Alles schließen, vorher Menu */
{                   /* Menu und RAM freigeben       */
  ClearMenuStrip(Window);
  FreeRaster(ram, 640,256);
  close_all();
}

/*--------------------------------------
  Die beiden folgenden Funktionen werden
  in Sitzung 10 ausgefüllt.
  --------------------------------------*/

void sysreq(char *text)
{
}

void do_datei(void)
{
}
```

Listing 2.9.1: Das Haupt-Programm »malen.c«

Workshop

Checkliste

1. Welche Aufgabe hat ein Requester, der sofort nach dem Öffnen des Windows erscheint?
2. Was müssen Sie tun, wenn Sie für eine Funktion mehrere Events derselben Klasse benötigen?
3. Warum ist die Funktion *WritePixel()* (Setzen eines Bildpunktes) so schlecht für das Malen geeignet?

Ideen für eigene Übungen

1. Die folgende Funktion zeichnet ein Rechteck von *x1,y1* (links oben) nach *x2,y2* (rechts unten):

```
void rechteck(SHORT x1, SHORT y1,
              SHORT x2, SHORT y2)
    {
      Move(rp, x1,y1);
      Draw(rp, x2, y1 );
      Draw(rp, x2, y2);
      Draw(rp, x1, y2);
      Draw(rp, x1, y1);
    }
```

Wenn Sie im *case* MOUSEMOVE anstatt *Draw(rp, mx, my)* nun *rechteck(x,y, mx, my)* einsetzen, ziehen Sie damit lauter Rechtecke. Wenn Sie danach in dieses Rechteck ein um ein Pixel (in jeder Richtung) blau gefülltes Rechteck einsetzen, müßte ein Rechteck übrig bleiben. Eine andere Methode heißt »zeichnen, löschen, zeichnen«. Zum Zeichnen setzen Sie den DrawModus mit *SetDrMd(rp, JAM1);* auf »normal«. Zum Löschen schreiben Sie *SetDrMd(rp, COMPLEMENT | JAM1).* Damit wird in der Komplementärfarbe gezeichnet, sprich, die vorhandene gelöscht.

2.10 Zehnte Sitzung: Daten von und zur Disk

Die Themen dieser Sitzung:

- Der Einsatz von Amiga-DOS
- File-Handles und Locks
- Zugriffsarten
- Planes und Farben
- Lesen und Schreiben von Bildern

Information

In dieser Sitzung geht es um Funktionen, die uns Amiga-DOS zur Verfügung stellt. Wesentlich ist natürlich die Frage, wie man Files (Dateien) auf die Diskette schreibt und wieder von dort liest. Ein gut geschriebenes Programm beläßt es aber nicht dabei, sondern stellt auch fest, ob noch genügend Platz auf der Diskette ist oder hilft dem Anwender bei der Suche nach einem File-Namen durch die Anzeige aller Files in einem Directory. Wir werden uns deshalb auch um diese Hilfsfunktionen kümmern müssen. Prinzipiell könnte man auch die Funktionen des Standard C hierfür einsetzen, nur ist das ein Umweg. Andererseits sind solche Programme nicht portabel, weshalb im Kapitel 3 auf dieses Thema noch eingegangen wird.

Locks für das Multitasking

Ein – jedenfalls beim Amiga – sehr wichtiger Begriff im Zusammenhang mit Files ist das »Lock«. Lock heißt auf deutsch Sperre, und eine Sperre ist ein Grundelement jedes Multitasking-Systems. Sie können sich sicherlich vorstellen, was passiert, wenn zwei Tasks (Programme) versuchen, gleichzeitig auf eine Diskette zu schreiben oder was ein File wert ist, wenn Sie daraus lesen und gleichzeitig ein anderer Task da Daten hineinschreibt. Genau das verhindern Locks. Es gibt zwei Arten von Locks, nämlich das »Exclusive Write Lock« und das »Shared Read Lock«. Nur der Inhaber des ersteren kann exklusiv auf die Datei schreiben. Ein »Shared

Read Lock« können sich mehrere Tasks teilen, sie können aber nur aus der gemeinsamen Datei lesen.

☞ Sie können mittels der Locks nicht auf Files schreiben oder aus ihnen lesen. Sie können (sollten) sich das Lock eines Files besorgen und daraus seinen Status ablesen, bevor Sie (mittels File-Handles) »zuschlagen«.

Ein Lock ist (natürlich) eine Struktur, wie sie Listing 10.1 zeigt.

```
 struct FileLock
{
 BPTR fl_Link;          /* Zeiger auf Nachfolger */
 LONG fl_Key;        /* Blocknummer auf der Disk */
 LONG fl_Access;              /* Zugriffs-Modus  */
 struct MsgPort *fl_Task; /* MsgPort der Tasks */
 BPTR fl_Volume      /* Zeiger auf Device-Liste */
};
```

Listing 2.10.1: Die Struktur eines Lock

Lesen und Schreiben von Files

Die DOS-Funktionen zum Lesen und Schreiben sind in der Syntax sehr an die entsprechenden Befehle der Hochsprachen angelehnt und deshalb sehr einfach zu handhaben. Zuerst wird ein File geöffnet mit

```
File_handle = Open(File_name, Modus);
```

Der Modus kann sein:

MODE_OLDFILE	File existiert.
MODE_NEWFILE	File wird neu angelegt oder ein vorhandenes File wird überschrieben.

Als Ergebnis erhält man ein File-Handle, das genaugenommen ein Zeiger auf eine Struktur vom Typ *FileHandle* ist. Sie können aber auch das File-Handle schlicht als ULONG deklarieren und die Adresse als File-Nummer auffassen. Wie auch immer, alle weiteren Funktionen verlangen als Parameter das File-Handle. Ist ein File offen, können Sie auf es mit *Read()* oder *Write()* zugreifen. Beide Funktionen haben die gleiche Syntax, nämlich:

```
Ist_Zahl = Write(File_handle, Puffer, Soll-Zahl);
Ist_Zahl = Read (File_handle, Puffer, Soll-Zahl);
```

Das heißt, es werden *Soll_Zahl* Bytes aus dem Puffer auf die Disk geschrieben, bzw. in ihn eingelesen. Der Puffer ist eine Variable, die Sie vor dem Schreiben mit Daten füllen müssen, beim Lesen werden die Daten von der Disk in den Puffer kopiert. Nach den Operationen steht in *Ist_Zahl*, wie viele Bytes tatsächlich geschrieben bzw. gelesen worden sind. *Ist_Zahl* ist besonders beim Lesen von Bedeutung. Da es in Amiga-DOS keine EOF-Funktion (End Of File) gibt, müssen Sie so lange lesen, bis *Ist_Zahl* null ist.

Praxis

Mit Listing 2.10.2 kommen wir zum ersten Programm, das die Aufgabe hat, ein File zu lesen und es anzuzeigen, also sozusagen ein Type-Befehl. Beachten Sie, daß Bytes gelesen werden, was immer sie auch bedeuten. Es gibt in Amiga-DOS keine Unterscheidung zwischen Text- und sonstigen Files.

```
/* dos_1.c */

#include <libraries/dos.h>
#include <exec/memory.h>

#define BUF_SIZE 256

void main(void)
{
 ULONG filehandle;
 UBYTE *buffer;
 int count;

 if ( (filehandle =  Open("dos_1.c",
       MODE_OLDFILE) ) == 0)
   exit(FALSE);

 buffer = (UBYTE *) AllocMem(BUF_SIZE,
          MEMF_CHIP | MEMF_CLEAR);
  do
   {
```

Listing 2.10.2: (Fortsetzung nächste Seite)

```
    count = Read(filehandle,buffer,BUF_SIZE);
    Write( Output(), buffer, count );
  }while(count); /* Solange nicht 0 Byte
                    gelesen werden */

 FreeMem(buffer,BUF_SIZE); /* RAM freigeben */
 Close(filehandle);        /* File schließen */
}
```

Listing 2.10.2: Das Lesen und Anzeigen eines Files

Als Quell-File wird schlicht der Name dieses Listings eingesetzt, das Sie deshalb als *dos_1.c* gespeichert haben sollten.

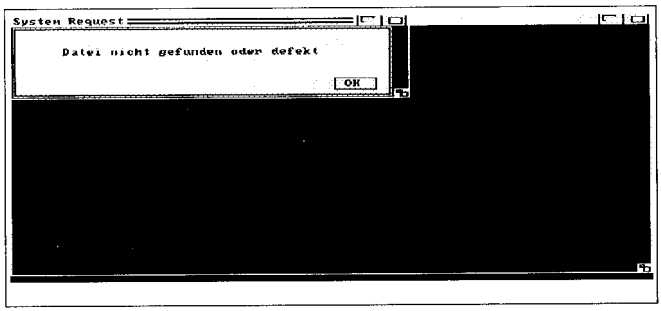


Bild 2.10.1: Das ist unser Ziel

Speicher dynamisch beschaffen

Wie schon geschildert, brauchen wir einen Puffer, der die Daten von der Disk aufnimmt. Aber anstatt einfach eine Variable *buffer[BUF_SIZE]* zu definieren, habe ich nur mit UBYTE ***buffer*** einen Zeiger angelegt. Der soll nun auf den Puffer zeigen, der wiederum BUF_SIZE (256) Byte groß sein muß. Diesen Speicherblock fordern wir beim Betriebssystem an, *Allokieren* nennt man das. Das geschieht mit

```
buffer = (UBYTE *) AllocMem(BUF_SIZE,
           MEMF_CHIP | MEMF_CLEAR);
```

Dazu gibt man die Größe des Speicherblocks und seinen Typ (oder seine Typen) an. Hier heißt MEMF_CHIP | MEMF_CLEAR nehme Chip-Memory oder was gerade noch frei (clear) ist. Danach müßte eigentlich noch das Ergebnis geprüft werden, doch ich nehme einfach an, daß die 256 Byte auf Ihrem Amiga noch frei sind (beim nächsten Beispiel prüfen wir).

Interessant ist schon der Grund, warum ich den Puffer dynamisch allokiert habe. Amiga-DOS wurde in BCPL geschrieben. Das ist eine C-ähnliche Sprache, die auf 32-Bit-Rechnern läuft. Zur Folge hat dies, daß BCPL-Adressen immer auf Langwort-Grenzen fallen. Der 68000er des Amiga ist aber eine Byte-Maschine, weshalb es für einen 68K-Compiler keinen Anlaß gibt, Daten auf Langwort-Grenzen zu justieren (Wortgrenzen sind üblich). Hält man sich in Amiga-DOS nicht an diese Langwort-Regel, geht das meistens auch gut, aber eben nur meistens. *AllocMem()* hingegen justiert garantiert auf »lang«.

Das eigentliche Lesen und die Ausgabe laufen mit

```
count = Read(filehandle,buffer,BUF_SIZE);
Write( Output(), buffer, count );
```

Readfile() versucht immer *BUF_SIZE* (256) Byte zu lesen, was bei den ersten Malen auch klappt. Dann sind es vielleicht nur noch zehn und beim nächstenmal null Byte. Dieser Wert steht in *count*. Die count-Bytes werden mit *Write()* geschrieben, nur nicht null Bytes, weil dann die Schleife abbricht. Doch wohin wird geschrieben? Nun, die Funktion *Output()* liefert das File-Handle des CLI-Windows, in dem Sie gerade sind. Damit erscheint die Ausgabe auf dem Schirm. Wenn Sie hingegen mit zum Beispiel *outhandle=Open("xxx", MODE_NEWFILE)* die Datei *xxx* öffnen und dann *Write(outhandle, buffer, count)* schreiben, gehen die Daten auf die Disk.

Status von Disketten ermitteln

Gehen die Daten wirklich auf die Disk? Es könnte ja sein, daß beim Schreiben die Diskette voll wird. Also formuliert man besser »Ist_Zahl = Write(Output(), buffer, count)« und meldet dann – wenn *Ist_Zahl == 0* ist – »Diskette voll«. Doch schön ist diese Methode auch nicht. Besser ist es, wenn man vorher prüft, ob noch genug Platz auf der Diskette ist. Wie man an diese Information herankommt, zeigt Listing 2.10.3.

```
/* dos_2.c */

#include <libraries/dos.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>
```

Listing 2.10.2: (Fortsetzung nächste Seite)

```
void main()
{
  struct FileLock *lock, *Lock();
  struct InfoData *infodata;
  if( !(lock = (struct FileLock *)
     AllocMem(sizeof(struct FileLock),
              MEMF_CHIP | MEMF_CLEAR)))
     exit();

 if ( !(infodata = (struct InfoData *)
      AllocMem(sizeof(struct InfoData),
               MEMF_CHIP | MEMF_CLEAR)))
     exit();

  if ( !(lock = Lock("DF0:",ACCESS_READ)))
    exit();

  if( !Info(lock,infodata) )
    exit();

  printf("Unit  %ld\n", infodata->id_UnitNumber);
  printf("Blöcke vorhanden: %ld\n",
         infodata->id_NumBlocks);
  printf("Blöcke belegt: %ld\n",
          infodata->id_NumBlocksUsed);

  UnLock(lock);
}
```

Listing 2.10.3: So erhält man den Disketten-Status

Das Listing 2.10.3 ist schon »Hohes C«, doch das knacken wir. Die benötigten Informationen liefert die Funktion *Info(lock,infodata)*. Die Argumente sind ein Lock, das müssen wir beschaffen und die Struktur *infodata*, die müssen wir bereitstellen. Diese Struktur zeigt Listing 2.10.4.

```
struct InfoData
{
 LONG id_NumSoftErrors, /* gefundene Disk-Fehler */
 id_UnitNumber,         /* DF0: == 0 */
 id_DiskState,          /* aktueller Status */
```

```
 id_NumBlocks,              /* vorhandene Blöcke */
 id_BlocksUsed,             /* belegte Blöcke */
 id_BytesPerBlock,
 id_DiskType;
 BSTR id_VolumeNode;        /*Zeiger auf BCPL-String
                             (Länge im ersten Byte) */
 LONG id_InUse
};
```

Listing 2.10.4: In der Info-Struktur stehen die Daten einer Disk

Für die Funktion *Lock()* brauchen wir auch ein *struct*, nämlich *lock* vom Typ *FileLock* (Listing 2.10.1). Sowohl *infodata* als auch *lock* sind aber nur Zeiger, und wie vorhin mit dem Puffer, allokieren wir jetzt den Speicher dafür mit *AllocMem()*. Das Besondere daran ist nur, daß wir jetzt die Blockgröße nicht kennen. Doch dafür bietet C die Funktion *sizeof()*, die als Argument eine Variable oder deren Typ akzeptiert. So ergibt *sizeof(struct FileLock)* die Größe einer FileLock-Struktur in Bytes. Folglich heißt der ganze Ausdruck bis dahin

```
lock = (struct FileLock *)
     AllocMem( sizeof(struct FileLock),
               MEMF_CHIP | MEMF_CLEAR )
```

Das Ergebnis ist null, wenn der Speicher nicht allokiert werden konnte. Folglich könnte man schreiben *if(lock == 0) exit()*. Logisch FALSE ist aber auch 0, ergo schreibt man kürzer *if(!lock)*, Sie erinnern sich, »!« ist der Nicht-Operator. Das klingt auch logischer, sozusagen wie »nicht allokiert«. Nun brauche ich aber diese Extra-Anweisung *if(!lock)* garnicht nicht, sondern kann auch gleich den ganzen Ausdruck in die if-Klammern setzen.

Nachdem nun auf diese Art auch Speicher für die Lock-Struktur allokiert wurde, beschaffen wir ein Lock für das Laufwerk DF0:. Ein Lock bekommt man ganz einfach mit *lock = Lock(Name, Modus)*. *lock* ist ein Zeiger auf eine Variable vom Typ *Lock*. *Name* ist der Name eines Files, eines Directorys oder einer Disk. Sperren kann man nur Files, die Locks für Directories und Disks braucht man, um Informationen über sie anfordern zu können. Der Modus kann ACCESS_READ (Lesen) oder ACCESS_WRITE (Schreiben) sein. Nun kommen wir endlich zur Funktion *Info()*, die das *Lock* und die infodata-Struktur als Argumente benötigt. In letzterer steht das Ergebnis gemäß Listing 2.10.4. Drei Komponenten davon werden mit *printf()* ausgedruckt.

Zugriff auf den FileInfoBlock

Nun kann auf der Disk noch alles OK sein, aber ein File macht Probleme. In diesem Fall müssen Sie in einer anderen Struktur nachsehen, nämlich im *FileInfoBlock* laut Listing 2.10.5.

```
struct FileInfoBlock
 {
 LONG fib_DiskKey;
 LONG fib_DirEntryType; /* Typ: >0 Directory,
                                 <0 File */
 char fib_FileName[108]; /* Name 0-terminiert */
                  /* woanders auf 30 begrenzt! */
 LONG fib_Protection; /* Bitmaske Bit 3-0 = rwxd */
 LONG fib_EntryType;
 LONG fib_Size;              /* Größe in Bytes */
 LONG fib_NumBlocks;         /* Belegte Blöcke */
 struct DateStamp fib_Date; /* Datum */
 char fib_Comment[80 ];      /* Kommentar-Feld */
 char padding[36]; /* Rest (Kommentar war früher
                      116 Byte lang*/
 };
```

Listing 2.10.5: Der FIB oder FileInfoBlock

Für *FileInfoBlock* schreibt man oft das Kürzel FIB. Die praktische Anwendung zeigt Listing 2.10.6. Der wesentliche Unterschied zu Listing 2.10.4 ist, daß anstatt DF0: ein File-Name eingesetzt wurde, und daß jetzt *f_info* vom Typ *FileInfoBlock* anstatt *InfoData* Verwendung findet. Außerdem wurde diese Struktur hier mit *Examine(lock, fib_zeiger);* gefüllt, einer Funktion, die wir gleich (in Listing 2.10.7) noch mehr brauchen werden.

```
/* dos_3.c */

#include <libraries/dos.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>

void main()
{
  struct FileLock *lock, *Lock();
```

```
    struct FileInfoBlock *f_info;

    if( !(lock = (struct FileLock *)
       AllocMem(sizeof(struct FileLock),
                MEMF_CHIP | MEMF_CLEAR)))
       exit();

    if ( !(f_info = (struct FileInfoBlock *)
        AllocMem(sizeof(struct FileInfoBlock),
                 MEMF_CHIP | MEMF_CLEAR)))
       exit();

    if ( !(lock = Lock("dos_3.c",ACCESS_READ)))
      exit();

    if( !Examine(lock,f_info) )
      exit();

    printf("Name : %s\n", f_info->fib_FileName);
    printf("Größe: %ld Bytes\n",f_info->fib_Size);

    UnLock(lock);
}
```

Listing 2.10.6: Zugriff auf den FIB

Nun fragen Sie mich, warum ich einen File-Namen eingebe, um ihn dann via FIB auszugeben? Da habe ich doch glatt zwei Antworten. 1. Man muß nicht mit

```
if ( !(lock = Lock("dos_3.c",ACCESS_READ)))
   exit();
```

aus dem Programm aussteigen, sondern kann anstatt *exit()* das Fehlen des Files beklagen oder allgemeiner gesagt, damit auf das Vorhandensein einer Datei prüfen. Antwort 2: Vielleicht haben Sie sich schon einmal gewundert, wieso auf »dir anton« das System »Anton« meldet? Der Grund: Wenn das File angelegt wird, wird der Name wie geschrieben eingetragen. Die dir-Routine unterscheidet aber bei der Suche nicht zwischen Groß- und Kleinbuchstaben.

Directory durchsuchen

Mit »dir« wären wir beim nächsten Thema und bei Listing 2.10.7. Das Programm basiert auf zwei Funktionen.

```
boolean = Examine(lock, fib_zeiger);
```

füllt einen FIB mit den Daten des Files von *lock*. Die Funktion gibt TRUE zurück, wenn sie Erfolg hatte. *lock* kann aber auch das Lock eines Directorys sein. Nun kommt die nächste Funktion zum Tragen.

```
boolean = ExNext(lock, fib_zeiger);
```

füllt den FIB des nächsten Files im Directory mit dessen Daten. Auf diese Art kann man sich sehr einfach durch ein Directory bewegen. Wenn Sie nun noch jedesmal in *fib_DirEntryType* nachsehen, ob es sich um ein File (<0) oder um ein Directory (>0) handelt, so können Sie auch sehr einfach an die Sub-Directories herankommen.

```
/* dos_4.c */

#include <libraries/dos.h>
#include <exec/memory.h>
#include <libraries/dosextens.h>

void print_it(info)
struct FileInfoBlock *info;
{
  printf("%30s", info->fib_FileName);
  printf("  %7d Bytes\n",info->fib_Size);
}

void main()
{
  struct FileLock *lock, *Lock();
  struct FileInfoBlock *f_info;

 if( !(lock = (struct FileLock *)
     AllocMem(sizeof(struct FileLock),
              MEMF_CHIP | MEMF_CLEAR)))
     exit();

 if ( !(f_info = (struct FileInfoBlock *)
```

```
        AllocMem(sizeof(struct FileInfoBlock),
                 MEMF_CHIP | MEMF_CLEAR)))
      exit();

  if ( !(lock = Lock("DF0:",ACCESS_READ)))
    exit();

  if( !Examine(lock,f_info) ) /* FIB des  /*
    exit();                   /* 1. Files /*

  print_it(f_info); /* Erstes FIB ausgeben */

  while(ExNext(lock,f_info)) /* Solange noch  */
    print_it(f_info);        /* weitere FIBs  */

  if (IoErr() != ERROR_NO_MORE_ENTRIES)
     printf("\nFehler im Directory\n");

  UnLock(lock);
}
```

Listing 2.10.7: So durchsucht man ein Directory

Wie Sie sehen, arbeitet unser Programm in der while-Schleife, so lange wie es einen FIB findet. Danach gibt es allerdings zwei Möglichkeiten, nämlich es gibt kein File mehr oder ein anderer Fehler ist aufgetreten. In diesem Fall – wenn *ExNext()* den Wert 0 hat – sollten Sie die Funktion *IoErr()* aufrufen. Wenn diese NO_MORE_ENTRIES ergibt, ist es ein normales Ende, andernfalls ist ein Lesefehler aufgetreten.

Sie können natürlich auch gleich ein Directory angeben, also anstatt DF0: zum Beispiel auch DH0:LC schreiben. Jokerzeichen sind allerdings nicht möglich. Wollen Sie dennoch nur Files listen, die beispielsweise mit ».c« enden, hilft ein Trick. Schreiben Sie vor *print_it(f_info)* die Zeile

```
if ( strstr(f_info->fib_FileName, ".c") )
```

Die *strstr()*-Funktion (string in string) gibt die Position des Suchstrings zurück, wenn der gefunden wurde, sonst null.

Execute anwenden

Man muß natürlich nicht – wie in Listings 2.10.7 – die File-Namen ausgeben, sondern kann mit den Namen weiterarbeiten. Wenn es Ihnen nur auf die Anzeige ankommt, und das Programm nur im CLI laufen soll, können Sie es auch billiger haben und zwar so:

```
#include <libraries/dos.h>
 void main()
{
 Execute("dir", NULL, Output() );
}
```

Die Funktion *Execute()* führt ein Programm aus, hier eines mit dem Namen *dir*. Alle CLI-Befehle sind ausführbare Programme. Außer dem Programm-Namen erwartet die Funktion zwei File-Handles, eines für den Input-File, eines für den Output-File. Ersteren habe ich hier auf NULL gesetzt, was erlaubt ist. Mit *Output()* trage ich das Handle des Standard-Outputs (normalerweise das CLI-Window) ein. Gibt man einen Input-File an, wird sein Inhalt als Befehlssequenz interpretiert, die nach dem Programm (hier: dir) ausgeführt wird. Für das Programm kann ich aber auch NULL einsetzen, dann werden sofort die Befehle im Input-File ausgeführt. Wissen Sie jetzt, worauf die Excecute-Routine des CLI basiert?

Schreiben und Lesen von Bildern

Nun zu unserer letzten Aufgabe. Im Malprogramm der neunten Sitzung (Listing 2.9.1) sind noch zwei Funktionen leer, die wir jetzt ausfüllen wollen; wie – zeigt Listing 2.10.8. Doch zuerst fügen Sie in Listing 2.9.1 noch diese Zeile nach den anderen *#includes* hinzu:

```
#include <libraries/dos.h>
```

Zuerst das Einfache: Die Funktion *sysreq(char *text)* wird mit einem String aufgerufen und zeigt diesen in einem Auto-Requester. Den kennen Sie schon aus der achten Sitzung, nur eines ist neu: Wenn man für den *Pos-Text* NULL einsetzt, hat der Requester nur noch ein Gadget. Für uns reicht das, denn der Anwender soll nur einen Text wie »ungültiger Datei-Name« sehen und dann auf OK klicken.

Damit ist auch schon der Zweck von *sysreq()* klar: Die Funktion wird aufgerufen, wenn in der Funktion *do_datei()* etwas schiefgeht. Diese Funktion soll unsere Bilder speichern oder laden. Wie schon mit Listing 2.10.2 gezeigt, muß man eine Datei zuerst mit *Open()* öffnen. Geht dabei etwas

schief, kommt unser Auto-Requester zum Einsatz. Die Menüwahl hatte bei »Laden« *dflag=1* und bei »Sichern« *dflag=2* gesetzt. Hiernach wird nun über das *case* entschieden, ob das Bild gelesen oder geschrieben werden soll.

Nehmen wir zuerst den Fall »Schreiben«. Ich wende dabei einen einfachen Trick an. Es wird nicht etwa das Window gespeichert, sondern der ganze Screen. Das geht sehr gut, weil unser Window die volle Schirmgröße hat und nicht verschoben werden kann (haben Sie diese Window-Flags gestrichen?). Genaugenommen liegt das daran, daß unser Window keinen eigenen Screen hat, sondern mit dem Workbench-Screen arbeitet. Damit ist auch der Rastport des Windows der des Screens. Im *RastPort* – unserer Variablen *rp* – gibt es einen Zeiger auf die BitMap-Struktur. In dieser wiederum befindet sich ein Array von Zeigern auf die Bit-Planes. Diese Planes schließlich sind nun (endlich) die RAM-Bereiche, in denen das Bild gehalten wird. Der Workbench-Screen hat zwei Planes (0 und 1). Jedes Bit in jeder Plane entspricht genau einem (und demselben) Bildpunkt. Die aus zwei Bit möglichen vier Kombinationen (00, 01, 10, 00) entsprechen vier Farbregister-Nummern. An eine Plane, zum Beispiel Plane 0, kommt man nun heran mit

```
rp->BitMap->Planes[0]
```

Um diesen RAM-Bereich auf ein File schreiben zu können, müssen wir ihn nur der Write-Funktion von Amiga-DOS als Argument übergeben. Da diese Funktion jedoch einen Zeiger auf einen Puffer vom Typ UBYTE erwartet, machen wir noch etwas »type casting to keep the compiler happy«. Das ergibt dann den schönen Ausdruck

```
Write(handle, (UBYTE *) rp->BitMap->Planes[0],
size);
```

»size« ist die Anzahl der zu schreibenden Bytes, die vorher mit *size = RASSIZE(640,256);* ermittelt wurde. Dafür hätte ich auch 20480 schreiben können, denn 640*256/8 ergibt das, aber mit dem Makro *RASSIZE(640,256)* sieht das natürlich viel eleganter aus. Mit noch einem *Write()* wird dann *rp->BitMap->Planes[1]* geschrieben, und das war's dann schon.

```
void sysreq(char *text)
{
  struct IntuiText BodyText, NegText;
  BOOL b;

  make_text( &BodyText, text, 50, 20 );
  make_text( &NegText, " OK ", 3, 3 );
  b = AutoRequest(NULL, &BodyText,NULL,&NegText,
                  NULL, NULL, 400,80 );
}


void do_datei(void)
{
  ULONG handle;
  int size;

  size = RASSIZE(640,256);

  switch(dflag)
  {
    case 1:
      if ((handle = Open(&info.Buffer[0],
                    MODE_OLDFILE)) == 0)
        {
          sysreq("Datei nicht gefunden oder defekt");
          return;
        }

      Read(handle, (UBYTE *) rp->BitMap->Planes[0],
                    size);
      Read(handle, (UBYTE *) rp->BitMap->Planes[1],
                    size);
      Close(handle);
    break;

  case 2:
     if ( (handle = Open(&info.Buffer[0],
          MODE_NEWFILE)) == 0 )
       {
```

```
            sysreq("Ungültiger Datei-Name");
            return;
          }

        Write(handle, (UBYTE *) rp->BitMap->Planes[0],
              size);
        Write(handle, (UBYTE *) rp->BitMap->Planes[1],
              size);
        Close(handle);
    break;
    }
}
```

Listing 2.10.8: Das Lesen und Schreiben von Bildern

Wenn Sie das Ganze perfekt erledigen wollen, müssen Sie die Werte der aktuellen Auflösung einsetzen. Dazu brauchen Sie:

rp->BitMap->BytesPerRow : X-Auflösung in Bytes
rp->BitMap->Rows : Y-Auflösung (direkt)
rp->BitMap->Depth : Tiefe = Anzahl Bitplanes

☞ Beachten Sie, daß die X-Auflösung in Bytes eingetragen ist, Sie also den Wert mit 8 malnehmen müssen. Ferner heißt eine Tiefe von 2, daß *es* eine *Planes[0]* und eine *Planes[1]* gibt.

Bliebe noch anzumerken, daß ich gar nichts zu den Farben sage, die Farbregler können ja beim Bildspeichern sonstwo stehen. Ich speichere jedoch das Bild ohne die Farbreglerstellung und hoffe, daß bis zum nächsten Laden niemand an den Reglern dreht. Wenn Sie das berücksichtigen wollen, müssen Sie die Ist-Farben mit *GetRGB4()* erfragen, sie mitspeichern und dann mit *SetRGB4()* setzen. Per Prinzip sieht das für alle 32 Register so aus (für die Normalauflösung braucht man nur die ersten 4):

```
for(i=0; i<=31; i++) /* für alle 32 Register */
  farbe[i] = GetRGB4(&Screen->ViewPort,i);
```

Die Farben sind in der Form »0x0RGB« (Rot, Grün, Blau) codiert. Jede Farbe belegt vier Bit, Halbbytes oder auch Nibbles genannt. Zum Setzen mit *SetRGB4(&Screen->ViewPort,i,R,G,B)* müssen Sie die Nibbles aus den Worten holen. Da ich dazu schon wahrhaft abenteuerliche arithmetische Übungen gesehen habe, hier eine C-gemäße Lösung.

Wenn »f« ein Farbwort ist, gilt:

```
R = f >> 8 & 0xF;
G = f >> 4 & 0xF;
B = f & 0xF;
```

Im Klartext: Das Rot-Nibble wird um acht Bit nach rechts geschoben und dann mit 0xF oder/und 0000000001111 verknüpft. Beim G-Nibble muß man nur um vier Bits schieben, und bei B steht das Nibble schon richtig, nur die anderen zwölf Bits müssen mittels der Maske auf null gesetzt werden.

Workshop

Checkliste

1. Wie prüfen Sie, wann beim Lesen aus einer Datei das File-Ende erreicht ist?
2. Warum sollte man DOS-Puffer immer mit *AllocMem()* allokieren?
3. In welcher Datenstruktur steht die Größe eines Files?

Ideen für eigene Übungen und Erweiterungen

1. Wandeln Sie Listing 2.10.2 so ab, daß das Programm immer nach 20 Zeilen anhält und auf einen Tastendruck wartet.
2. In der Funktion *do_datei()* wird nicht geprüft, ob die Funktionen *Read()* und *Write()* 0 zurückgeben. Bauen Sie diese Tests ein und geben Sie für den Fehlerfall mittels *sysreq()* Meldungen aus. Wenn Sie ganz gut sein wollen, speichern Sie die Ist-Größe mit ab und vergleichen sie dann mit der gelesenen. Sie können auch noch ein Kennwort mit speichern und das zurücklesen, um zu verhindern, daß eine falsche Datei eingelesen wird.
3. Wandeln Sie Listing 2.10.7 so ab, daß alle Files mit der Endung ».IMG« (empfohlener Extender für Bilder) angezeigt werden. Bauen Sie das dann in das Malprogramm ein. Dazu verlängern Sie das Menü »Projekt« um den Punkt »Dateien zeigen«. Beachten Sie, daß Sie die Texte mit der Funktion *print()* (in Listing 2.7.5) ausgeben müssen und dabei jeder Text eine neue x-y-Position braucht.

Kapitel 3: Know-how

Dieses Kapitel ist primär in zwei Teile gegliedert. Der erste Teil beschäftigt sich mit dem Thema »Fehlerbeseitigung«, der zweite behandelt einige Besonderheiten der Compiler und hilft mit einer Sammlung von Kniffen, Tips und anderen Spezialitäten, Ihr C-Wissen weiter zu verfeinern.

3.1 Häufige Fehler und Lösungswege

Sie werden es kaum glauben, doch Profis machen genauso viele Fehler wie Einsteiger. Der kleine Unterschied: Die Profis brauchen dafür ein paar Programmzeilen mehr und – das ist es – sie haben alle Fehler schon einmal gemacht und daraus gelernt. Dieses Know-how soll Ihnen die folgende Aufzählung möglicher Fehlerquellen vermitteln.

Ein Gleichheitszeichen vergessen

Der Zuweisungsoperator heißt =, der Gleichheitsoperator hingegen ==. Verwechselt man das, kann das Programm falsche Ergebnisse bringen oder abstürzen. Ein harmloses Beispiel:

```
int i = 123;
if (i = 10 )
  puts("i = 10");
```

Das Programm wird »i = 10« ausgeben, obwohl *i* 123 ist. Wenn eine Zuweisung als Ausdruck eingesetzt wird, bekommt dieser den Wert der Zuweisung, hier also 10. Das ist nicht null, und somit für C logisch TRUE.

Die if-Bedingung ist damit erfüllt. Schlimm wird es, wenn man das mit Zeigern macht, also zum Beispiel *If (Window = NULL)* schreibt. Damit erhält *Window* den Wert 0, also eine sicherlich falsche Adresse. Beim nächsten Zugriff darauf kracht es.

Klammern vergessen

In der while-Schleife sollen so lange Zeichen in die Variable *ch* eingelesen werden, bis die Escape-Taste gedrückt wird, doch leider funktioniert das so nicht:

```
while ( ch = getch() != '\x1B' )
  printf("%c",ch);
```

Der Grund ist, daß der Ungleichoperator (!=) eine höhere Priorität als der Zuweisungsoperator hat. Folglich wird zuerst verglichen und erst danach mit *ch = getch()* das Zeichen eingelesen. Die Lösung ist ganz einfach. Umgehen Sie mit der Klammerung die automatische Priorisierung. Hier also so:

```
while ( (ch = getch() ) != '\x1B' )
```

☞ Kleine Zusatzregel: Setzen Sie im Zweifelsfall immer Klammern, eher zu viele und überflüssige als zu wenig. Der Code wird dadurch nicht schlechter, doch die Lesbarkeit des Programms wird besser.

Array-Index außerhalb der Dimension

Besonders bei der Übertragung von Basic- oder Pascal-Programmen begeht man häufig folgenden Fehler bei der Übersetzung von z.B. *for i=1 to 4*:

```
int ar[4] = {10, 20, 30, 40};
int i;
for(i=1; i<=4; i++)
  printf("%d\n", ar[i] );
```

Die Schleife läuft von 1 bis 4, ein C-Array beginnt aber mit dem Index 0. Folglich muß die Schleife heißen

```
for(i=0; i<4; i++)
```

Mit dem falschen Index bekommen Sie noch nicht einmal eine Fehlermeldung, weil C nicht auf Array-Grenzen achtet. Peinlich wird die Sache in

der Umkehr, also zum Beispiel mit der Zuweisung *a[55] = 13;*. In diesem Fall schreiben Sie irgendwo in den Speicher, der Schaden kann groß sein.

Falscher Index bei mehrdimensionalen Arrays

Auch bei der Übertragung von Basic- oder Pascal-Programmen begeht man häufig folgenden Fehler. Aus *DIM a(3,5)* wird in C

```
int a[3][5];
```

Nun kann man aber im Eifer des Gefechtes leicht

```
printf("%d", a[1,2] );
```

schreiben und bekommt keine Fehlermeldung, sondern nur ein falsches Ergebnis. Der Grund ist, daß es korrekt

```
printf("%d", a[1][2] );
```

heißen muß. Eine Fehlermeldung gibt es nicht, weil C in der Form *a[1,2]* den Komma-Operator sieht, daher den Ausdruck von links nach rechts entwickelt und schließlich dafür *a[2]* einsetzt. Das jedoch ist leider nur die Adresse des Array-Elements.

Fehler im Umgang mit Strings

Zur Erinnerung: C-Strings müssen am Ende ein 0-Byte, genauer ein Null-Zeichen haben. Ein Hauptfehler ist das vergessene Null-Byte bzw. die Tatsache, daß ein String immer wenigstens um 1 größer dimensioniert werden muß als die maximale Textlänge. Im Falle von

```
char s1[5], s2[5];

main() {
  strcpy(s1,"Hallo");
  strcpy(s2,"Welt");
  printf("%s%s\n",s1,s2);
}
```

kopiert *strcpy()* den String *Hallo* plus ein 0-Zeichen, also sechs Zeichen, in den Bereich von *s1*, für den der Compiler nur 5 Byte reserviert hatte. Folglich liegt das 0-Byte außerhalb des Array *Meldung*, hier im Bereich von *s2*. Das nächste *strcpy()* kopiert den String *Welt* dorthin, womit das *W* wieder das Null-Byte überschreibt. *s1* wird gedruckt, bis ein Null-Byte

gefunden wird, und das steht erst am Ende von *s2*. Dann wird *s2* gedruckt. Das Ergebnis ist *HalloWeltWelt*. Wenn Sie das Beispiel nachvollziehen wollen, beachten Sie: Die Strings müssen außerhalb von *main()* deklariert sein, womit sie statisch werden (immer auf derselben Adresse stehen). Außerdem sollten Sie mit den Längen experimentieren, weil verschiedene Compiler die Strings unterschiedlich auf Wort- oder Langwortgrenzen ablegen können.

Eine andere Falle ist diese:

```
char s1[] = "Hallo";
char *ptr;
ptr = malloc( strlen(s1) );
strcpy(ptr, s1);
```

Diesmal ist der Fehler gut versteckt. Die strln()-Funktion ergibt die Stringlänge ohne das Null-Byte. Dafür alloziert *malloc()* Speicher. Das anschließende *strcpy hängt* aber wieder das Null-Byte an. Korrekt muß die Zeile also lauten

```
ptr = malloc( strlen(s1) + 1 );
```

Probleme mit Zeigern

Vier Fehler sind sehr beliebt, nämlich nicht initialisierte Zeiger, falsch initialisierte Zeiger, Zeiger vom falschen Typ und »dangling pointers« (dangle = wacklig, lose). Beginnen wir mit den nicht initialisierten Zeigern. Sie schreiben in einem Programm ganz ordentlich

```
struct RastPort *rp
/* und etwas später */
Move(rp, x,y);
```

Leider hatten Sie vergessen, mit *rp=Window->RPort* dem Zeiger *rp* auch eine Adresse zuzuweisen. Die Move()-Funktion schreibt deshalb mitten in den Systemspeicher: der Crash ist vorprogrammiert. Auch auf diese Art kann man zu einem initialisierten Zeiger kommen, jedenfalls, wenn man den Warnlevel des Compilers hochsetzt:

```
int wert = 10;
int *ptr;
*ptr = &Wert;
```

**ptr* meint nicht mehr den Zeiger *ptr*, sondern die Adresse, auf die er zeigt. Dahin wandert auch die Adresse von *Wert*, nur *ptr* selbst ist immer noch nicht initialisiert.

Im Abschnitt über Strings hatte ich geschrieben

```
ptr = malloc( strlen(s1) );
strcpy(ptr, s1);
```

Auch das ist recht gefährlich, denn ich weiß gar nicht, ob *malloc()* den Speicher noch beschaffen konnte. In diesem Fall gibt es NULL zurück, was meistens auch die Adresse 0 ist. Korrekt schreibt man also das Stückchen Code so:

```
char s1[] = "Hallo";
char *ptr;
ptr = malloc( strlen(s1) + 1 );
if (ptr == NULL)
  exit(1);
else
  strcpy(ptr, s1);
```

Eine böse Falle kann die Funktion *scanf()* sein. Bekanntlich muß man *scanf("%d", &i)* schreiben, also die Adresse der Variablen angeben, damit *scanf()* sein Ergebnis da eintragen kann. Vergißt man das &-Zeichen, passieren gleich zwei Fehler. Die Variable bekommt keinen Wert, ihr Inhalt ist zufällig, und – viel schlimmer – *scanf()* betrachtet diese Zufallszahl als Adresse und schreibt auf diese.

Ein »dangling pointer« entsteht auf vielfältige Weise. Ich brauche bloß nach *ptr = malloc(...)* irgendwann

```
free(ptr);
```

zu schreiben, und schon kann das System diesen Speicherbereich anderweitig vergeben. Der Pointer zeigt immer noch dahin, aber leider auch aus Sicht Ihres Programms in die Wüste. Ob ein nochmaliger Zugriff klappt, bleibt Zufall. Eine weitere Möglichkeit, einen »dangling pointer« zu erzeugen, ist diese:

```
char *error(int i)
{
  char *c_ptr;
  .
  .
  .
  c_ptr = "Hallo";
  return c_ptr;
}
```

Die Funktion ist syntaktisch OK, nur logisch nicht. *c_ptr* ist eine lokale Variable, die nur so lange existiert, wie die Funktion läuft. Der Return-Wert der Funktion ist zwar noch der richtige Zeiger, nur zeigt er auf eine Variable, die es nach dem Funktionsende gar nicht mehr gibt.

Zeiger auf sich selbst

Sie haben zum Beispiel in der Menü-Struktur *menu[1]* einen Zeiger auf den Nachfolger *menu[2]*. Lassen Sie den aber nicht auf das nächste Menü zeigen, sondern auf sich selbst, hier *menu[1]*, entsteht ein schwer zu findender Fehler, weil das Programm sich aufhängt, sobald es *SetMenuStrip()* aufruft.

3.2 Details zum Aztec-Compiler und Linker

Der C-Compiler *cc* wird mit

```
cc -Option1 -Option2 -Option...  name
```

aufgerufen. Im Beispiel *cc –a –r4 –sf* name bedeuten:

cc	Compiler-Aufruf
–a	Erzeuge Assembler-Quelltext
–r4	Nehme A4 für Registervariable
–sf	Optimiere for-Schleifen
name	Name des Quellfiles

Die Optionen müssen nicht sein. *cc test* kompiliert *test.c*. Der Extender ».c« wird automatisch angehängt, wenn er nicht angegeben wurde. Der Aztec-Compiler glänzt mit derart vielen Optionen, daß ich hier nur die wichtigsten aufführen kann und ansonsten auf das sehr umfangreiche Handbuch verweisen muß.

–3	Akzeptiert Optionen der Version 3.6
–a	Erzeugt (zusätzlich) Assembler-Quelltext
–bs	Erzeugt Debug-Informationen
–fa	Fließkommazahlen nach Amiga-IEEE
–ff	Fließkommazahlen nach »Fast Motorola«
–fm	Fließkommazahlen nach Manx (Aztec)
–f8	Fließkommazahlen für 68881
–hi	Siehe 3.6, vorkompilierte H-Files
–ho	Siehe 3.6, vorkompilierte H-Files
–i Pfad	Directory, wo *cc* Include-Files suchen soll
–k	Kompiliere nach K&R-Standard (UNX V7)
–mc	Code für das große Speichermodell
–md	Daten für das große Speichermodell
–pp	Alle *char* werden zu *unsigned char*
–ps	*int* als 16 Bit anstatt 32
–qp	Erzeugt Prototypen für alle nicht statischen Funktionen
–qs	Erzeugt Prototypen für alle statischen Funktionen
–sa	2-Assembler-Lauf für diverse Optimierungen
–sf	Optimiert for-Schleifen
–ss	Mehrfach deklarierte Strings nur einmal ablegen
–wa	Warnt, wenn Funktionsaufruf nicht nach Prototyp
–ws	Keine Warnings mehr (gefährlich)
–wu	Warnung, wenn lokale deklarierte Variable nicht genutzt (nützlich).

Der Aztec-Linker

Der Linker *ln* wird mit

```
ln Option1 Option2 Option...  name1 name2 name...
```

aufgerufen. Im Beispiel *ln +cd name –lc* bedeuten:

ln	Linker-Aufruf
+cd	Daten ins Chip-Memory
name	Name des Objektfiles
–lc	Library *c.lib* einbinden

Die Form *ln name –lc* ist eine Kurzform von *ln name.o c.lib*

Die Optionen müssen nicht sein. *ln test* –lc linkt *test.o* zu einem ausführbaren Programm. Der Extender ».o« wird automatisch angehängt, wenn er nicht angegeben wurde. Der Aztec-Linker bietet folgende Optionen:

+a	Justiert auf Langwortgrenzen
+cb	BSS-Daten (nicht initialisierte) ins Chip-Memory
+cc	Code geht ins Chip-Memory
+cd	Daten ins Chip-Memory
+ccdb	Alles ins Chip-Memory
+f	Programm geht ins Fast-Memory
–fname	Liest Linker-Kommando aus Datei *name*
–g	Erzeugt File für Aztec-Quellcode-Debugger
+l	Alle folgenden Namen sind Libraries, wirkt wie ein Schalter bis zum nächsten +l
–lname	Library *name.lib* einbinden
–m	Warnt nicht, wenn Library-Symbole überschrieben werden.
–o name	Name des Programm-Files, sonst wie Name des ersten Input-Files.
o[i]	Plaziert das folgende Modul in das Segment *i*. Wird *i* nicht angegeben, wird das erste Segment genommen.
–q	Schaltet –g wieder aus
+q	Schaltet Modulanzeige beim Linken aus
–t	Erzeugt Symbol-Tabelle (lesbarer Text)
–w	Symboltabelle für ROM-Wack-Debugger erzeugen
–v	Zeigt die wichtigsten Link-Ergebnisse wie Code- und Datengröße an.

3.3 Details zum Lattice-Compiler und Linker

Der Lattice-Compiler glänzt mit derart vielen Optionen, daß ich hier nur die wichtigsten aufführen kann und ansonsten auf das sehr umfangreiche Handbuch verweisen muß. Der Compiler *lc* wird mit

```
lc -Option1 -Option2 -Option... name1 name2 name...
```

aufgerufen. Im Beispiel *lc –L –ad test* bedeuten:

–L	Nach dem Kompilieren den Linker aufrufen
–ad	Daten ins Chip-Memory
test	test.c kompilieren

Weil der Linker BLINK schon mit der –L-Option aufgerufen wird, und dessen Optionen hier mitgeschildert werden, verzichte ich auf die Beschreibung von BLINK.

Der Extender ».c« wird automatisch angehängt, wenn er nicht angegeben wurde. Im Beispiel *lc –L –ad test* waren es zwei Optionen, es können aber auch mehr sein. Bei sich widersprechenden Optionen warnt *lc*. Hier nun die Liste der wichtigsten Optionen:

–ad	Daten gehen ins Chip-Memory
–ac	Code geht ins Chip-Memory
–ab	BSS-Daten (nicht initialisierte) ins Chip-Memory
–acdb	Alles ins Chip-Memory
–b1	Kurze Adressierung. Reicht nur für 64 Kbyte Daten, –b1 ist voreingestellt
–b0	Lange Adressierung (32 Bit)
–ca	Volle ANSI-Kompatibilität
–cf	Compiler moniert fehlende Funktionsprototypen
–cm	Erlaubt mehrfache Zeichenkonstasten wie 'ab'
–co	Kompatibel mit früheren Versionen
–cs	Mehrfach deklarierte Strings nur einmal ablegen
–cu	Alle *char* werden zu *unsigned char*
–cusf	Bester Code und bestes Error-Checking
–d0	Keine Debug-Information im Code
–d5	Volle Debug-Information im Code
–H	Siehe 3.6, vorkompilierte H-Files
–fl	Fließkommazahlen nach Lattice
–ff	Fließkommazahlen nach »Fast Motorola«
–fi	Fließkommazahlen nach Amiga-IEEE
–f8	Fließkommazahlen für 68881
–iPfad	Directory, wo *lc* Include-Files suchen soll
–L	Compiler ruft Linker auf, der *lc.lib* und *amiga.lib* einbindet.
–L+Name.lib	Bindet *Name.lib* zusätzlich ein
–Lc	Erzeugt »small code«
–Lm	Bindet *lcm.lib* für Fließkommazahlen
–l	Alle Daten auf Langwortgrenzen justieren
–O	Schaltet den globalen Code-Optimierer ein
–ph	Siehe 3.6, vorkompilierte H-Files

3.4 (Überlebens-)Regeln und Tips für Programmierer

Es folgen gleich viele (Überlebens-)Regeln und Tips für Programmierer – doch halt! Ich meine, bevor ich die Regeln so einfach aufzähle, sollte ich wenigstens andeuten, warum man sie braucht, sprich, ein paar Grundlagen bringen.

Multitasking macht Exec

Zuständig für das berühmte Multitasking des Amiga ist Exec. Exec ist die Abkürzung von Executive, was im Amerikanischen soviel wie Chef (leitender Angestellter) bedeutet. Exec ist an sich auch nur ein Prozeß, wie zum Beispiel das CLI, der aber immer (so lange der Amiga eingeschaltet ist) läuft. Exec selbst ist das Multitasking-System des Amiga. So ein System hat die Aufgabe, die Ressourcen eines Systems auf verschiedene Tasks (Programme) zu verteilen. Ressourcen sind die CPU (der 68000), der Hauptspeicher und die Peripherie-Geräte. Unteilbare Ressourcen wie die CPU werden beim Amiga nach einem Prioritäten-Verfahren vergeben. Zuerst wird der Task mit der höheren Priorität abgearbeitet, dann der nächstniedere. Haben mehrere Tasks die gleiche Priorität, werden sie nach einem Zeitscheibenverfahren (Time-Sharing) in Intervallen bearbeitet. Durch Hardware-Interrupts bekommt Exec die Sache immer wieder in den Griff, auch wenn ein Task seine Tätigkeit nicht beenden möchte.

Ein Task hat drei Zustände, nämlich laufend, nicht laufend und wartend. Mit Rücksicht auf andere Tasks sollte ein Task möglichst oft in den Zustand *wartend* (auf Eingabe) gesetzt werden, das aber via *Wait()* und nicht etwa nach der unfeinen Polling-Methode (siehe Regel 9).

Der Zugriff auf System-Routinen

Um von absoluten Adressen unabhängig zu sein, arbeiten die meisten Betriebssysteme nach diesem Schema: Alle Unterprogramme erhalten eine Nummer, Funktionsnummern genannt. Im Betriebssystem steht eine Tabelle, in der notiert ist, welche Adresse zu jeder Funktionsnummer gehört. Das Betriebssystem hat nun eine Routine, deren Adresse sich nie ändert. Das ist der Dispatcher. Um ein Unterprogramm aufzurufen, übergibt man dem Dispatcher die Funktionsnummer. Dieser berechnet danach (und mit Hilfe der Tabelle) die Adresse der Routine und ruft sie auf. Der Amiga macht das etwas raffinierter und damit zukunftssicherer. Der

Nachteil der Standard-Methode ist nämlich, daß man sehr schlecht neue Routinen hinzufügen kann (Tabelle steht im ROM). Beim Amiga stehen die Tabellen im ROM oder RAM oder auf der Diskette. Die Tabellen sind ein Teil der sogenannten Libraries (Bibliotheken).

Libraries: Schlüssel zum Amiga

Eine Library ist, vereinfacht ausgedrückt, eine Sammlung von Unterprogrammen mit einer zugehörigen Tabelle (je Unterprogramm ein Eintrag). Für jeden Zweck (zum Beispiel DOS, Intuition, Grafik) gibt es eine eigene Library. Will man eine Funktion einer Library benutzen, muß man die Library mit *OpenLibrary()* öffnen. »Gemanagt« wird das Ganze vom sogenannten Library-Manager (ein Teil von Exec). Der Manager weiß, ob sich eine Library schon im ROM oder RAM befindet. Wenn nicht, versucht er, die Library von der Diskette zu laden. Klappt das nicht (Library ist nicht auf der Diskette oder der Speicher ist schon voll), gibt er 0 als Adresse zurück.

Der Umstand hat noch einen Grund: Wir haben ein Multitasking-System, was auch heißt, daß quasi gleichzeitig verschiedene Tasks (Programme) eine Library benutzen können. Der erste Task wird die Library notfalls von der Diskette in den RAM laden (genau: das Laden veranlassen). Öffnen weitere Tasks dieselbe Library, wird der Manager nur noch die Adresse an diese Tasks melden und sich merken, daß dieser Task die Library (auch) braucht. Daraus folgt: Eine Library darf erst wieder aus dem Speicher gelöscht werden, wenn der letzte Task gesagt hat, daß er sie nicht mehr braucht. Dafür gibt es die Funktion *CloseLibrary*. Jeder Task (also jedes Programm, das Sie schreiben) muß deshalb alle Libraries, die er geöffnet hat, auch wieder schließen. Andernfalls könnte bald der Speicher knapp werden.

Regel 1: Es gibt nur eine absolute Adresse

Man sieht immer wieder Listings, die auf absolute Adressen zugreifen, was übrigens in C ganz einfach ist. Man muß nur einer Zeigervariablen die Adresse (als Zahl) zuweisen. Doch nur eine einzige Adresse, nämlich die 4, ist erlaubt; im Listing 3.5.5 sehen Sie eine praktische Anwendung. Diese Adresse 4 ist die *_SysBase* und dort liegt der Zeiger auf die Basis von »Exec«. Alle weiteren Adressen sind über definierte Offsets und System-Funktionen zu ermitteln. Gegebenenfalls erhält man damit die Zeiger auf Datenstrukturen, die wiederum Zeiger auf andere Datenstrukturen halten. Die doppelte Indirektion (**) ist zwar oft die Folge solcher

Wege, aber diese Methode ist die einzig zulässige. Gehen Sie davon aus, daß Programme mit absoluten Adressen nur auf Ihrem Amiga und nur mit der einen Kickstart- und Workbench-Version laufen.

Regel 2: Funktions-Ergebnisse testen

Testen Sie generell die Ergebnisse aller Funktionen, die nicht ausdrücklich als *void* deklariert sind. Sie müssen davon ausgehen, daß Ihr Programm in Konkurrenz zu vielen anderen läuft (Multitasking), und somit durchaus nicht klar ist, ob noch der Speicher für beispielsweise eine neue Window- oder Screen-Struktur frei ist. Gehen Sie davon aus, daß fast jede Aktion Speicher kostet. Im Extremfall müssen Sie sogar unterstellen, daß die paar Bytes fehlen, die für das Öffnen einer ROM-Library benötigt werden.

Regel 3: Speicher wieder freigeben

Geben Sie auf jeden Fall allokierten Speicher wieder frei und nicht nur den mit *malloc()* beschafften. Schließen Sie auch alle Screens, Windows und die Libraries. Beachten Sie diese Regel besonders im Fehlerfall. Hier passiert es häufig, daß man Speicher für einige Datenstrukturen beschafft hat, dann etwas schiefgeht, und vor dem Abbruch des Programms eben nicht der Speicher für die bisher allokierten Strukturen freigegeben wird. Natürlich kann es vorkommen, daß Ihr Programm laut Regel 4 festgestellt hatte, daß der Speicher ausreicht, nun loslegt, und es dann »mitten drin« doch nicht reicht. Der Grund ist schlicht, daß sich ein anderer Task den Speicher inzwischen »geschnappt« hat. Nun haben Sie aber derweil eine unbekannte Anzahl von Speicherblöcken für Images, Sprites und sonstiges reserviert. Wie geben Sie diese wieder frei?

Die Lösung bietet Intuition mit seiner Funktion *AllocRemember()*. Ein Beispiel dazu finden Sie im Abschnitt 3.5.3. Diese Funktion notiert alle erfolgreichen Allokierungen in einer gemeinsamen Liste, einer Struktur vom Typ *Remember*, die Sie zur Verfügung stellen müssen, genauer: – siehe auch Listing 3.5.1 – einen Zeiger darauf. Nehmen wir an, die Variable heißt **rem*, dann können Sie mit diesem einen Befehl

```
FreeRemember(&rem, TRUE)
```

alle Speicherblöcke auf einmal freigeben. Beachten Sie unbedingt das »TRUE«, denn – es ist kaum zu glauben – setzen Sie hier »FALSE« ein, wird nur die Liste gelöscht und nicht die Speicherblöcke.

Regel 4: Frühzeitig testen

Kein Anwender findet es schön, wenn ein Programm erst anläuft und dann »mitten drin« aussteigt. Es gilt zwar auch für diesen Fall die Regel »informieren Sie den Anwender über den Grund des Abbruchs«, doch am besten prüfen Sie gleich zu Programmbeginn, ob noch genügend Speicher frei ist und die sonstigen Voraussetzungen für diese Applikation erfüllt sind. Wenn nicht, nennen Sie dem User die Gründe und die möglichen Abhilfemaßnahmen. Wenn dennoch im Programmlauf Fehler auftreten und alle Stricke reißen, müssen Sie sogar mit einem Dead-End-Alert aussteigen (siehe Sitzung 8), also nach einer Guru-Meldung neu booten lassen. Still aussteigen und den Anwender mit einem anderen Programm abstürzen zu lassen, ist nicht die feine englische Art.

Regel 5: Der Ressource-Manager sind Sie

Im Gegensatz zu den ganz großen Multitasking-Systemen müssen Sie beim Amiga selbst etwas – wenn auch sehr wenig – tun, wenn es um den Zugriff auf System-Ressourcen geht. Wenn zwei Tasks auf Ressourcen wie ein Laufwerk gleichzeitig zugreifen wollen, gibt es Konflikte. Um diese zu vermeiden, müssen Sie diese Aktion einkleiden in

```
forbid()      /* Multitasking aus */
/* Aktion */
permit()      /* Multitasking an  */
```

Natürlich sollte das *forbid()* nur so kurz wie möglich wirken, weil, solange alle anderen Tasks keine Chance haben, doch viel wichtiger ist, daß man das *permit()* nicht vergißt.

Regel 6: Chip-Daten ins Chip-RAM

Die Custom-Chips des Amiga (Blitter, Copper usw.) können Hardwarebedingt nur auf die ersten 512 Kbyte – das sogenannte Chip-Memory – zugreifen. Folglich müssen auch alle Datenstrukturen, die von diesen Bausteinen genutzt werden, in diesem Bereich liegen. Hat der Amiga aber mehr als 512 Kbyte RAM, wird ein Programm in aller Regel in den Erweiterungsspeicher – das sogenannte »Fast Memory« – geladen. »Fast« (schnell) heißt das übrigens, weil »da oben« die CPU nicht mehr Taktzyklen an die Custom-Chips abgeben muß. Mit dem Programm wandern auch die Daten in das »Fast Memory«, sind also für die Custom-Chips unerreichbar. Günstigenfalls führt das zu einem verschwundenen Maus-

zeiger und ähnlichen Effekten, doch auch Abstürze sind drin. Die möglichen Lösungen werden in den Abschnitten 3.5.2 und 3.5.3 beschrieben.

Regel 7: Das Rad nur einmal erfinden

Daß man häufig benutzte Routinen in Include-Files hält und die dann immer wieder einzieht, ist eine uralte Maßnahme, doch beim Amiga ist sie besonders wichtig. Wenn man sich nämlich bei den vielen komplexen Datenstrukturen nur einmal ein wenig verhaut, reicht das oft schon für einen bildschönen und schwer zu findenden Absturz. Besonders kritisch sind übrigens als Konstanten definierte Datenstrukturen. Vergißt man da einmal eine Komponente oder nimmt den falschen Typ, »gurut« es garantiert. Deshalb: Alles was läuft, gleich in einen Include-File packen und den im nächsten Programm einsetzen. Ein ganz nützliches Exemplar dieser Gattung ist übrigens unser so oft genutztes *stdwindow.h*.

Regel 8: Beachten Sie die »Postvorschriften«

Bei der deutschen Bundespost gibt es sehr genaue Vorschriften über den Transport und die Behandlung von Nachrichten. Beim Amiga gilt ähnliches, kommunizieren doch alle Tasks – die eigenen und die des Betriebssystems – mittels Messages (elektronischer Briefe), die sie sich gegenseitig schicken. Dazu noch einige Details: Jede Eingabe beim Amiga läuft über das sogenannte Input-Device in den Input-Stream. Letzterer ist ein Strom von Daten, genauer ein Puffer-Bereich. Wenn der Anwender eine Taste drückt, die Maus bewegt oder eine Maus-Taste betätigt, generiert das Input-Device daraus ein Input-Event (Input-Ereignis). Aus dem Input-Event wird eine Message (Nachricht), die das Ereignis beschreibt.

Der Input-Stream kann nun von jedem Task abgefragt werden. Diese Abfrage ist nicht ganz einfach, na sagen wir, ziemlich umständlich zu programmieren. Deshalb sollten wir auch diesen Job Intuition überlassen, da sind nämlich die passenden Routinen schon eingebaut. Dabei funktioniert Intuition als eine Art Filter, das nur die uns interessierenden Events durchläßt. Intuition sorgt automatisch dafür, daß die Events immer an den derzeit aktiven Task (an das aktive Window) weitergeleitet werden. Die Events gelangen (neben dem Console-Device) in den IDCMP (Intuition Direct Communication Message Port). Der IDCMP ermöglicht den Empfang aller Ereignisse im »Rohformat«. Bei vielen Ereignissen stört das nicht, das Window-Close-Event oder die Mauskoordinaten zum Beispiel können gar nicht besser vorliegen.

Die Abfrage des IDCMP in einem Programm ist an sich ganz einfach. Wir erhalten eine Message, wenn ein Event eingetreten ist. Im Class-Feld des Message-Ports steht dann ein Langwort, in dem ein Bit gesetzt ist, das dem Ereignis-Typ entspricht. Dieses Bit entspricht genau dem des IDCMP-Flags, das Sie vorher gesetzt hatten. Haben wir also beim Öffnen des Windows CLOSEWINDOW gesetzt, könnten wir theoretisch nun folgendes tun:

```
msg = GetMsg(Window->UserPort) /* Lese  Message */
  if( msg->Class == CLOSEWINDOW ) .....
```

Praktisch dürfen wir aber so nicht verfahren, denn zwei Regeln sollten Sie immer im Auge behalten, nämlich:

- Nach GetMsg() muß ReplyMsg() folgen
- Nach ReplyMsg() sind die Daten im Message-Port ungültig

Daraus folgt mindestens dieser Ablauf:

1. Message mit *GetMsg()* holen
2. Daten aus Message-Port in andere Variable kopieren
3. Mit *ReplyMsg()* den Erhalt der Message quittieren
4. Auf Message reagieren
5. Weiter bei 1.

Das »mindestens« möchte ich sehr betonen, weil ein so geschriebenes Programm gegen eine Grundregel des Multitasking verstößt. Jedes Programm sollte, wenn es auf ein Ereignis wartet, das auch sagen. Praktisch geschieht dies mit

```
Wait(1L << Window->UserPort->mp_SigBit)
```

Damit meldet sich der Task sozusagen mit der Meldung ab »Ich gehe jetzt schlafen, weckt mich, wenn ein Ereignis für mich anliegt«. Praktisch wird damit der Task von Exec aus der Liste der aktiven Tasks auf die Liste der wartenden Tasks versetzt, womit anderen Tasks mehr Zeit zugeteilt werden kann. Tritt das Event ein, wird der Task automatisch wieder aktiviert.

Die falsche Methode wäre das sogenannte Polling. Hierzu fragt das Programm in einer Schleife ständig den Message-Port ab, ob eine Message anliegt. Natürlich vergehen bis dahin aus CPU-Sicht Ewigkeiten, die Antwort kann also durchaus 9999mal Nein und dann einmal Ja lauten. Leider

blockiert diese Abfragerei CPU-Zeit, die den anderen Tasks fehlt. Solche Programme erkennt man am fehlenden *Wait()* und einer Programmzeile dieser Art

```
while( msg = GetMsg(Window->UserPort) == 0 )
  ;
```

Beachten Sie das Semikolon. Manchmal steht es noch auf derselben Zeile wie *while()*, was dann auch noch schlechter Programmierstil ist.

Regel 9: Denken Sie an FGO

Die drei Buchstaben FGO stehen für Funktionalität, Geschwindigkeit und Oberfläche. Die Regel stammt aus der Macintosh-Welt – woher auch sonst? Mit diesem Computer begann der Siegeszug der grafisch orientierten Bedienoberflächen und auch der Lernprozeß der Programmierer. Damit wären wir beim Thema. Wer von einem mauslosen Computer auf den Amiga umsteigt, ist natürlich fasziniert von den Möglichkeiten dieser Grafikmaschine und legt für sein neues Programm erst einmal eine Benutzeroberfläche hin, die so intuitiv bedienbar ist, daß jedes Handbuch überflüssig wird. Das erste Ziel eines solchen GUI (Graphic User Interface), nämlich die leichte Erlernbarkeit eines Programms, wurde also erreicht.

Doch die Anwender sind unverschämte Leute. Kaum, daß sie mit dem Programm umgehen können und seine Funktionen beherrschen, verlangen sie auch schon, daß diese Funktionen möglichst schnell ausgeführt werden. Ein Testbericht der Art, daß Ihr Super-Wordprozessor zum Ersetzen aller »e« durch »xyz« vier Sekunden braucht und die Konkurrenz nur derer zwei, kann schon das große Aus sein. Schauen Sie sich um (und auf sich selbst), und Sie werden merken, daß die schnellsten Programme einer Klasse immer auch die beliebtesten sind. Häufig werden sogar zugunsten der hohen Arbeitsgeschwindigkeit Mängel im GUI in Kauf genommen.

Woraus folgt: Die leichte Erlernbarkeit ist zwar ein edles Ziel, sie darf aber nicht dazu führen, daß das Programm für den fortgeschrittenen Anwender zu langsam wird. Langsam ist ein Programm in diesem Sinne auch dann, wenn keine Abkürzungen vorhanden sind. Sie müssen sich das vor Augen führen.

Anstatt

- Drücken der rechten Maustaste
- + Anfahren des Menüs
- + Wahl von »Sichern«
- + Warten auf File-Dialog
- + Scrollen durch die Listbox
- + Wahl des File-Namens aus der Listbox
- + Klick auf OK

tippt der »advanced user« doch lieber Ctrl + s <name> Return. Kleine Anmerkung dazu: Auch wenn wissenschaftlich erwiesen ist daß der kurze Wechsel von der Tastatur zur Maus und zurück schneller ist als so manches Tastenkürzel, hilft Ihnen das gar nichts. Wenn der User den subjektiven Eindruck hat, auf die falsche Art schneller zu sein, dann geben Sie ihm seine Tasten für alle Lebenslagen.

Um wieder auf das FGO zu kommen: Kümmern Sie sich zuerst um das F, also um alle nötigen und – noch besser – um möglichst viele Funktionen. Mit der Funktionsvielfalt steigt nämlich die Zahl der potentiellen User (Käufer). Dann sehen Sie zu, daß diese Funktionen mit der höchstmöglichen Geschwindigkeit ausgeführt werden. Erst an dritter Stelle kommt das O. Das heißt nun nicht, daß Ihr GUI den Charme einer Schreibmaschine haben darf, doch die Oberfläche ist OK, wenn sie dem Intuition-Standard entspricht. Opfern Sie keine Minute für Gags im GUI, so lange F und G nicht stimmen. Gerade die kommerziellen Anwender und die Power-User sehen über manchen Mangel in der Oberfläche hinweg, wenn sie von der Funktionsvielfalt und dem Tempo begeistert sind.

3.5 Kniffe, Tips und Spezialitäten

In diesem Abschnitt finden Sie einige Dinge, mit denen Sie das Malprogramm erweitern können – ein neuer Mauszeiger und Grafiktexte gehören dazu –, aber auch einige Tricks für C im allgemeinen und die Amiga-C-Compiler im besonderen.

3.5.1 Workbench-Programme mit Icons

Um unser Malprogramm – und überhaupt jedes Intuition-Programm – von der Workbench aus starten zu können, benötigt es nur noch ein Icon. Damit ein Icon sichtbar wird, muß es a) vorhanden sein (logisch) und b) in einem Directory stehen, das selbst ein Icon hat, sprich, als Schublade sichtbar ist. Am einfachsten stellen Sie eine solche Schublade her, indem Sie auf der Workbench die Schublade »Empty« duplizieren. Sie können aber auch im CLI einfach tippen

```
copy empty.info test.info
```

Kehren Sie nun zur Workbench zurück, sehen Sie das neue Icon nicht. Schließen Sie dann das Disk-Fenster und öffnen es wieder. Nun ziehen Sie die Empty-Schublade etwas weg, und die Test-Schublade wird sichtbar.

Jetzt brauchen wir ein Programm-Icon. Dazu nehmen Sie am besten auch ein vorhandenes Icon, allerdings nicht jedes ist geeignet. Der Amiga kennt verschiedene Typen von Icons. Welche das sind und was sie für eine Bedeutung haben, erfahren Sie automatisch, wenn Sie den Icon-Editor starten. Für uns ist es wichtig, zu wissen, daß Programme vom Typ TOOL sein müssen. Geeignet ist zum Beispiel IconED selbst. Nehmen wir an, Sie haben das Directory (die Schublade) »test« schon erstellt und unser Programm hieße »program«. Dann kopieren Sie zuerst das Programm mit

```
copy program :test/program
```

Nun kopieren Sie ein Icon dazu (IconEd steckt im Tools-Ordner)

```
copy :tools/iconed.info :test/program.info
```

Nun sollten Sie auf der Workbench in der Schublade »test« ein Icon finden, das aussieht wie das von IconEd, aber den Titel »program« zeigt. Das können Sie nun getrost anklicken, »program« wird starten. Wenn Sie jetzt Ihrem Icon ein eigenes Aussehen verpassen wollen, rufen Sie IconEd auf. Im Disk-Menü wählen Sie *Load* und tippen dann in den Text-Requester

```
:test/program
```

sprich, immer den vollen Pfadnamen ein. Das Editieren ist simpel und im Prinzip selbsterklärend. Probieren Sie einfach die verschiedenen Menü-Punkte aus. Wichtig zu wissen ist: Um ein Icon zeichnen/ändern zu kön-

nen, müssen Sie immer das Menü *Color* anwählen und daraus die passende Farbe. Radieren können Sie mit der Hintergrundfarbe. Gezeichnet/radiert wird mit der Maus. Die linke Taste drückt den Stift auf das »Papier«. Sie kennen das Prinzip? Richtig, unser Malprogramm arbeitet genauso.

Sie können aber auch auf den Kopiervorgang ganz verzichten und ein Icon selbst im Editor erstellen. Sie müssen dann nur im Save-Requester den korrekten Namen eingeben, in unserem Beispiel also wieder *:test/program*. Ansonsten keine Sorge. Es können zwar die unmöglichsten Icons entstehen, aber editiert wird immer nur das Info-File. Ihrem Programm passiert nichts.

3.5.2 Chip-Memory per eigenem Mauszeiger

Um einen eigenen Mauszeiger zu erzeugen, braucht man die Funktion *SetPointer()*. Mit *ClearPointer()* kann man wieder auf den Standardzeiger (den Pfeil) zurückschalten. Doch niemand zwingt Sie, in einem Programm nur einen Pointer (Mauszeiger) einzusetzen. Es können mehrere sein, und ständig umschalten dürfen Sie auch. Genau das wollen wir hier auch üben.

⚠ **Doch bevor Sie nun loslegen, zuerst eine Warnung. Ein Pointer ist ein Sprite, und Sprites funktionieren nur im Chip-Memory (in den unteren 512 Kbyte Ihres Amiga).**

Wenn Sie einen Amiga mit mehr als 512 Kbyte haben, müssen Sie etwas tun. Am einfachsten ist es, vor diesem Programm die Utility »NoFastMem« (im System-Ordner) aufzurufen. Sie können aber auch einen Compiler-Switch setzen, der die Daten in das Chip-Memory zwingt. Das hat zwar den Nachteil, daß dann alle Daten in das an sich immer knappe Chip-Memory gelangen, doch damit wollen wir beginnen. Sollte Ihnen diese Lösung nicht zusagen, dann warten Sie bis zum nächsten Abschnitt.

Listing 3.5.1 hat diese Aufgabe: Wenn der Mauszeiger in der Arbeitsfläche des Fensters ist, soll er ein X sein, kommt er hingegen in die Titelleiste, soll er die Pfeilform annehmen.

☞ Beachten Sie unbedingt die Compiler/Linker-Kommandos zu Beginn des Listings. Genau damit müssen Sie nachher das Programm kompilieren bzw. linken.

```
/*
 *   mp.c  Mouse-Pointer setzen
 *
 *   Compiler/Linker-Kommandos:
 *   --------------------------
 *
 *   Aztec:      cc mp
 *   -----       ln +cd mp -lc
 *
 *   Lattice:    lc -L -ad mp
 *   -------
*/

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;
struct IntuiMessage *msg;
ULONG class;
SHORT my;

#include "stdwindow.h"

USHORT Maus[] =
        {
          0x0000, 0x0000,

          0x0830, 0x0C10,
          0x0460, 0x0620,
          0x02C0, 0x0340,
          0x0180, 0x0180,

          0x0380, 0x01C0,
          0x0640, 0x0260,
          0x0C20, 0x0430,
          0x1810, 0x0818,
```

```
            0x0000, 0x0000
        };
void main()
{
    open_libs();

    Window = (struct Window *) open_window(
          20,20,600,170,
          "Data im Chip-Memory per Compiler-Option",
          WINDOWCLOSE | REPORTMOUSE | ACTIVATE,
          CLOSEWINDOW | MOUSEMOVE, NULL);
    if (Window == NULL) exit(FALSE);

    SetPointer(Window, &Maus, 8, 8, -4, -4);

    do{
        Wait(1L << Window->UserPort->mp_SigBit);
        while( msg = (struct IntuiMessage *)
                  GetMsg(Window->UserPort) )
        {
          class = msg->Class;
          my = msg->MouseY;
          ReplyMsg( msg );
        }
        if ( my < 11 )
           ClearPointer(Window);
        else
     SetPointer(Window, &Maus, 8, 8, -4, -4);
    }while(class != CLOSEWINDOW);

    close_all();
}
```

Listing 3.5.1 Ein Window mit eigenem Mauszeiger

Ein Pointer kann bis zu 16 Pixel breit und beliebig hoch sein. Sein Image wird zeilenweise in einem Array, hier *Maus[]*, abgelegt, wobei dann jedes gesetzte Bit in diesen Worten einem gesetzten Punkt in der Bit-Plane entspricht. Arbeitet man wie hier mit zwei Bit-Planes (vier Farben), so müssen die Zeilen in der Folge

```
Zeile 1 für Plane 1
Zeile 1 für Plane 2
Zeile 2 für Plane 1
Zeile 2 für Plane 2
usw.
```

abgelegt sein. Über diese sinnige Folge habe ich mich übrigens auch schon beim Atari ST gewundert. Nun denn, spielen wir mit. Ansonsten gibt es ja auch Sprite-Editoren, die solche Arrays kreieren. Ist das eigentliche Muster fertig, müssen Sie es noch mit je zwei Null-Wörtern am Anfang und Ende einrahmen. Mein Sprite ist ein schlichtes X mit Einsen in beiden Planes, das ergibt die Farbe 3 (binär 11). Nun könnten Sie auch schon *SetPointer()* aufrufen. In der Zeile

```
SetPointer(Window, &Maus, 8, 8, -4, -4);
```

ist *Window* unser Window-Zeiger, *&Maus* zeigt auf die Sprite-Daten und nun kommt's: Die beiden Achten beschreiben die Breite und Höhe des Pointers, das ist ja noch einsichtig, aber die beiden Vieren sind das X- bzw. Y-Offset vom »hot spot«. Dieser heiße Punkt ist der Punkt, der über einem Ziel sein muß, wenn Sie die Maustaste drücken. »0,0« heißt linke obere Ecke des Pointer-Rechtecks, »–4,–4« hingegen vier nach rechts, vier nach unten, hier also genau die Mitte. Warum das negative Vorzeichen sein muß, weiß ich nicht, aber es muß sein.

Der neue Pointer wird wirksam, sobald er im Window ist, und das Window aktiv ist. Klicken Sie einfach Ihr Window an, das X müßte erscheinen. Sobald Sie außerhalb des Windows klicken, entsteht wieder der alte Pointer. Nun gehört aber die Titelleiste noch zum Window, und ich wollte, daß sich schon hier die Zeigerform ändert. Genau das machen die Zeilen

```
if ( my < 11 )
          ClearPointer(Window);
      else
   SetPointer(Window, &Maus, 8, 8, -4, -4);
```

my ist die Mausposition, unter y=11 beginnt die Titelleiste. Ich habe das hier in eine vereinfachte Form der Event-Schleife gepackt. Wenn Sie das Ganze in das Malprogramm einbauen wollen, sollten Sie diese Zeilen dem Case MOUSEMOVE zuordnen.

3.5.3 Daten ins Chip-Memory legen mit AllocRemember()

Die bisherigen Lösungen zum Thema Chip-Memory haben Sie nicht überzeugt? Sie haben recht, mich auch nicht so ganz. Wenn ich schon viel Geld für die Speichererweiterung bezahlt habe, will ich sie nicht mit »NoFastMem« wieder abschalten. Und ein ganzes Programm will ich auch nicht in die unteren 512 Kbyte legen, denn erstens ist da nicht viel Platz, und zweitens sind die Programme da unten langsamer. Das mit dem Compiler-Switch geht zwar schon, doch wenn man viele Daten im Programm hat, ist das auch nicht so gut. Doch die Lösung ist ganz einfach. Man darf halt nur die Daten in das Chip-Memory legen, die da unbedingt sein müssen, zum Beispiel die Pointer-Strukturen. Wie man das macht, zeigt Listing 3.5.2.

```
/* cmem.c
   Mauszeiger-Array im Chip-Memory
   ------------------------------- */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>

#define MEMF_CHIP 2

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct Window *Window;

#include "stdwindow.h"

USHORT Maus[] =
{
  0x0000, 0x0000, 0x0830, 0x0C10, 0x0460, 0x0620,
  0x02C0, 0x0340, 0x0180, 0x0180, 0x0380, 0x01C0,
  0x0640, 0x0260, 0x0C20, 0x0430, 0x1810, 0x0818,
  0x0000, 0x0000
};
```

Listing 3.5.2: (Fortsetzung nächste Seite)

```
void main()
{
    struct Remember *RememberKey;
    UBYTE i;

    struct
     {
       USHORT m[20];
     } *MP, *AllocRemember();

    open_libs();
    Window = (struct Window *) open_window(
             20,20,400,170,
             "Data im Chip-Memory per Programm",
             WINDOWCLOSE | ACTIVATE, CLOSEWINDOW,
             NULL );
    if (Window == NULL) exit(FALSE);

    RememberKey = NULL;
    MP =  AllocRemember(&RememberKey, sizeof(Maus),
                        MEMF_CHIP );
    for (i=0; i<20; i++)
      MP->m[i] =  Maus[i];
    SetPointer(Window, &MP->m[0], 8, 8, -4, -4);
    Wait(1L << Window->UserPort->mp_SigBit);
    FreeRemember(&RememberKey, TRUE);
    close_all();
}
```

Listing 3.5.2: So bringt man Daten in das Chip-Memory

Das Programm generiert denselben Mauszeiger, wie das vorherige, nur wird dessen Datenstruktur in das Chip-Memory gezwungen. Kern der Übung ist diese Zeile:

```
MP = AllocRemember(&RememberKey, sizeof(Maus),
      MEMF_CHIP );
```

Es handelt sich dabei um eine erweiterte *malloc()*-Funktion. Der *RememberKey* zeigt auf die Remember-Struktur, in der Intuition die Speicherreservierungen verwaltet. Vor dem ersten Aufruf muß dieser Zeiger

mit NULL geladen werden. Das zweite Argument ist die gewünschte Größe (wie bei *malloc()*), das dritte aber ist das, was wir wollen, nämlich eine Vorgabe für den Zielbereich. Wenn Sie hier MEMF_CHIP eintragen, wird der Speicher im Chip-Memory allokiert. Die Konstante habe ich hier mit *#define* definiert, weil es mir widerstrebte, nur wegen einer Konstanten einen ganzen Include-File (hier *memory.h*) zu laden.

Normalerweise beschafft man sich Speicher für ganze Strukturen, weshalb ich in diesem Beispiel auch eine *struct* angelegt habe, nur um einmal das »Wie« aufzuzeigen. Da steht zwar jetzt nur der Platz für den Array *m[20]* drin, aber Sie können ja auch noch die übrigen Sprite-Daten dazu packen, so wie es sich eigentlich auch gehört.

MP (wie Mouse Pointer) ist nun der Zeiger auf diese Struktur im Chip-Memory. Der Einfachheit halber erspare ich mir nun die 20 Zuweisungen, sondern kopiere einfach die Daten aus der schon vorhandenen Struktur mittels der for-Schleife.

Zum Schluß muß man den Speicher wieder freigeben, und das geschieht mit

```
FreeRemember(&RememberKey, TRUE);
```

Das Argument TRUE besagt, daß sowohl die Listen-Knoten frei gegeben werden sollen, die auf die einzelnen Speicherblöcke zeigen (vorausgesetzt, *AllocRemember* wurde mehrfach aufgerufen), als auch die Speicherblöcke selbst. Setzen Sie dieses Argument auf FALSE, wird nur die Liste gelöscht, eine gewagte Aktion, wie ich meine, also bleiben Sie »true«.

3.5.4 Zeichensätze, Größen und Stilarten

Wir hatten zwar *IntuiText* schon oft genutzt, aber diesmal geht es zur Sache. Deshalb schauen wir uns dieses *struct* jetzt genauer an.

```
struct IntuiText
 {
 UBYTE FrontPen, BackPen;
 UBYTE DrawMode;
 SHORT LeftEdge, TopEdge;
 struct TextAttr *ITextFont;
 UBYTE *IText; struct IntuiText *NextText;
 }
```

Habe ich eine Variable dieses Typs, zum Beispiel mit dem Namen *MyText*, kann ich Text ausgeben mit

```
PrintIText(rp, &MyText, x,y);
```

rp ist dabei wieder der RastPort. Etwas verwirrend sind *x* und *y*, denn mit *LeftEdge/TopEdge* in der Struktur wurde die linke obere Ecke des Textbeginns auch schon festgelegt. Die Lösung: *x* wird auf *LeftEdge* und *y* auf *TopEdge* addiert, weshalb man diese beiden Werte in der Struktur meistens auf 0 setzt. Interessanter ist deshalb der Rest der Struktur. *FrontPen* gibt die Farbnummer des Vordergrundes an, *BackPen* die für den Hintergrund. Ob *BackPen* wirksam wird, hängt von *DrawMode* ab. Ist dieser JAM1, überschreibt der Text den Hintergrund nur im Bereich der Zeichen. Ist *DrawMode* hingegen JAM2, wird der Hintergrund im Bereich der Zeichenmatrix mit *BackPen* beschrieben. Ich kann damit zum Beispiel auf dem blauen Hintergrund des Schirms einen orangen Streifen erzeugen, in dem mit schwarzer Schrift ein Text steht. Beide Modi können noch mit COMPLEMENT oder INVERSVID »verodert« werden. COMPLEMENT komplementiert die vom Text »getroffenen« Bits (aus 0 wird 1, aus 1 wird 0). Im Ergebnis ist das eine andere Farbe. INVERSVID (invers Video) muß zusammen mit JAM1 oder JAM2 gebraucht werden.

JAM1 | INVERSVID	Ergibt transparente Zeichen, umrahmt von der Hintergrundfarbe.
JAM2 | INVERSVID	Läßt die Zeichen in der Hintergrundfarbe erscheinen.

Ganz wichtig ist nun *ITextFont*. Das ist ein Zeiger auf einen Font (Zeichensatz). Ist dieser Zeiger NULL, wird der (langweilige) Standardzeichensatz genutzt. *ITextFont* kann aber auch einen Font zeigen, der von der Diskette geladen wurde. Listing 3.5.3 zeigt, wie man zu solchen Fonts kommt.

```
/* fonts.c */

#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
ULONG  DiskfontBase;
```

Listing 3.5.2: (Fortsetzung folgende Seiten)

```
struct Window *Window;
struct RastPort *rp;
struct TextFont *font_rom, *font[2];
struct TextAttr ta_rom, ta[2];
struct IntuiText intuitext;

#include "stdwindow.h"

/* Funktion zum Öffnen eines Disk-Fonts
   ---------------------------------- */
open_font(struct TextAttr *ta, char name*,
          SHORT size, SHORT style, SHORT flags,
          struct TextFont  *font)
{
  ta->ta_Name = (UBYTE *) name;
  ta->ta_YSize = size;
  ta->ta_Style = style;
  ta->ta_Flags = flags;
  if (!(font = (struct TextFont *)
  OpenDiskFont(ta)))
    exit(FALSE);
}

make_ftext(struct IntuiText *name, char *text,
           struct TextAttr *ta)
{
  name->FrontPen = 1;
  name->BackPen  = 0;
  name->DrawMode = JAM1;
  name->LeftEdge = 0;
  name->TopEdge  = 0;
  name->ITextFont = ta;
  name->IText = (UBYTE *) text;
  name->NextText = NULL;
}

void main()
{
  if( !(DiskfontBase = OpenLibrary
```

Listing 3.5.2: (Fortsetzung folgende Seiten)

```
 ("diskfont.library",0)) )
  exit(1);
open_libs();
Window = (struct Window *)
          open_window(0,0,640,256,"Titel",
          WINDOWCLOSE | WINDOWSIZING | ACTIVATE,
          CLOSEWINDOW | VANILLAKEY,
          NULL
         );
if (Window == NULL) exit(1);
rp = Window->RPort;
SetAPen(rp, 1L);

/* Zuerst eine Übung mit dem ROM-Font
   ---------------------------------- */
ta_rom.ta_Name = (UBYTE *) "topaz.font";
ta_rom.ta_YSize = TOPAZ_SIXTY;
ta_rom.ta_Style = 0;
ta_rom.ta_Flags = FPF_ROMFONT;
if ( font_rom = (struct TextFont *)
     OpenFont(&ta_rom) )
 {
   SetFont(rp, font_rom);
   Move(rp,50, 50);
   Text(rp,"Das ist Topaz 60", 16);
   ta_rom.ta_YSize = TOPAZ_EIGHTY;
   Text(rp," und das ist Topaz 80",21);
   CloseFont(font_rom);
 }

 /* Nun das Öffnen und Nutzen von Disk-Fonts
    ---------------------------------------- */
 open_font( &ta[0],  "sapphire.font", 19, 0, 0,
            font[0]);
 make_ftext(&intuitext, "Das ist Saphire",
  &ta[0]);
 PrintIText(rp, &intuitext, 50,70);
```

Listing 3.5.2: (Fortsetzung folgende Seiten)

```
    open_font( &ta[1],  "emerald.font",  20, 0, 0,
               font[1]);
  make_ftext(&intuitext, "Das ist Emerald", &ta[1]);
    PrintIText(rp, &intuitext, 50,90);

    /* Ist der Font offen, kann man z.B. die Stilart
       ändern.
    */
    ta[1].ta_Style = 1; /* unterstrichen */
    intuitext.IText = (UBYTE *) "Unterstrichen";
    PrintIText(rp, &intuitext, 250,90);

    /* Ist der Font-Zeiger NULL, wird der ROM-Font
       eingesetzt
    */
    make_ftext(&intuitext, "Das ist wieder der ROM-
               Font", NULL);
    PrintIText(rp, &intuitext, 50,120);

    Wait(1L << Window->UserPort->mp_SigBit);

    CloseFont( font[0] ); /* Fonts schließen! */
    CloseFont( font[1] );
    CloseLibrary(DiskfontBase); /* Library auch */
    close_all();
}
```

Listing 3.5.3: So öffnet man Fonts

Kern der Übung ist eine Struktur namens *TextAttr* (Text-Attribute). Wie das Listing zeigt, muß man mindestens zwei Werte in diese Struktur eintragen, nämlich den Font-Namen und seine Größe *YSize* (Höhe). In der Wahl der Höhe sind Sie nicht frei, sondern an den Bestand gebunden. Schauen Sie sich einmal das Font-Directory auf Ihrer Workbench-Disk an, die Auswahl ist groß. Zum Beispiel finden Sie unter *fonts/times* die Fonts mit den Namen 11, 13, 15, 18 und 24. Die Namen sind auch die Größen. Flags können Sie außer acht lassen, Style hingegen erlaubt Ihnen folgende Stilarten:

0 normal
1 unterstrichen
2 fett
4 italic (kursiv)
8 Breitschrift

Sie sehen, es handelt sich wieder um Bits, Sie können also diese Zahlen verodern oder addieren, um die Stilarten zu kombinieren. Das System macht aber nicht bei jedem Font alles mit, und manche wilde Kombination ist auch hübsch häßlich. Probieren Sie es aus.

Außerdem gibt es zwei ROM-Fonts, beide haben den Namen Topaz. Der Unterschied liegt in der Höhe. Die vordefinierten Konstanten TOPAZ_SIXTY und TOPAZ_EIGTHY bedeuten nicht 60 und 80 hoch, sondern bezeichnen die Fonts, die bei 60 bzw. 80 Zeichen pro Zeile eingesetzt werden. Den Unterschied sehen Sie kaum, dem hinter den beiden Konstanten stecken die Höhen 8 bzw. 9. Auf jeden Fall können Sie auch einen im ROM befindlichen Font mit *OpenFont()* öffnen, wie hier gezeigt, und erhalten dann einen Zeiger auf eine Struktur vom Typ *TextFont*. An sich brauchen Sie für alles Weitere nur den Zeiger. Wenn Sie an der Struktur selbst drehen wollen (vorsichtig!), schauen Sie sie sich einmal an (steht in *graphics/text.h*).

Um an die interessanteren Diskfonts heranzukommen, muß man nur den richtigen Namen in die TextAttr-Struktur einsetzen und dann anstatt *OpenFont() OpenDiskfont()* nehmen. Natürlich muß der Font auf der Diskette sein. Für den Fall, daß dem nicht so ist: ohne jede Meldung einfach mit *exit()* auszusteigen, ist nicht die feine Art. Hier sollten Sie noch einen Auto-Requester einbauen (siehe Sitzung 8).

Texteingabe für das Malprogramm

Um im Malprogramm auch Texte einzugeben, gehen Sie wie folgt vor: Zuerst öffnen Sie alle Fonts, die Sie einsetzen möchten. Ich habe deshalb schon mit *ta[2]* und **font[2]* Arrays vorgesehen, wo Sie nur noch die Zahlen anpassen müssen. Dann bieten Sie in einem Menü die Fonts an und bekommen von dort die Item-Nummer *i* von 0 bis zum Beispiel 3 geliefert. Jetzt müssen Sie nur noch in einer Zeile wie

```
make_ftext(&intuitext, "Das ist Emerald", &ta[1]);
```

anstatt *&ta[1] &ta[i]* schreiben. Außerdem werden Sie für *"Das ist Emerald"* eine Stringvariable einsetzen, die muß nur ein Zeichen aufnehmen können. Warum das so ist, zeigt Listing 3.5.4.

```
strvoid do_text(int dx)
{
  char s[] = " ";
  SHORT x,y;

  Titel("Klicke auf Textstart und tippe bis
Return");
  do{
      GetEvents();
    }while (code != SELECTDOWN);
  x = mx; y = my;
  Move(rp, x, y);

  do{
      GetEvents();
      if (class == VANILLAKEY && code != 13 )
        {
          s[0] = code;
          Text(rp, s, 1);
          x += dx; /* Zeichenabstand */
          Move(rp, x,y);
        }
    }while(code != 13);

  Titel("Standard-Modus");
}
```

Listing 3.5.4: Texteingabe im Malprogramm

Sie bieten einen Menüpunkt »Texteingabe« an, und wie bei *do_rechteck()* wird jetzt *do_text()* aktiv. Sie klicken auch zuerst auf den Startpunkt, nur danach ändert sich etwas. Die do-Schleife endet mit *while(code != 13);*, d.h., die Schleife läuft so lange, wie nicht die Return-Taste (Code 13) gedrückt wird. Innerhalb dieser Schleife wird mit »if (class == VANILLAKEY ;&& code != 13)« sichergestellt, daß die folgenden Aktionen nur bei einem Tastatur-Event (VANILLAKEY) ausgeführt werden, aber nicht mehr beim Code 13. Was nun kommt, hat folgenden Sinn: Die Funktion *Text()* braucht als Argument einen String, wir haben aber nur ein Zeichen. Folglich wird im String *s* das erste und einzige Zeichen (s[0])

durch den Code ersetzt. Die Funktion *Text()* wurde hier nur der Einfachheit halber eingesetzt. Sie müssen dafür praktisch

```
make_ftext(&intuitext, s, &ta[i]);
PrintIText(rp, &intuitext, x,y);
```

schreiben. Sie können auch das *make_text()* aus *menu.h* verwenden, müssen dann aber für *left* und *top* noch zwei Nullen übergeben. Bliebe noch zu erwähnen, daß Sie die Funktion mit zum Beispiel *do_text(10)* aufrufen müssen. Das ist nämlich der Parameter *dx* für den Zeichenabstand, der vom Font und der Größe abhängig ist. Wenn Sie auch noch die Schriftgrößen und die Stilarten variieren wollen – noch ein paar Menüs – kein Problem. Übergeben Sie *do_text()* auch noch diese Argumente und setzen dann *ta[i].ta_YSize* und *ta[i].ta_Style* auf die passenden Werte.

3.5.5 So schreibt man kompakte Programme

Es gibt viele Tricks, kompakte Programme zu schreiben, womit ich allerdings die Code-Größe meine und nicht Bandwurmzeilen im Quelltext. Letztere wirken sich übrigens oft nachteilig auf die Code-Größe aus. Ich will hier auch nicht Bytes zählen, nach dem Motto, daß *i+=3* kürzeren Code ergibt als *i=i+3*, denn solche Dinge werden zunehmend auch schon von den Compilern erkannt. Binsenweisheiten wie *puts()* ist kürzer als *printf()* möchte ich auch nicht wiederholen, also was bleibt noch?

_main() anstatt main() bei Lattice

In reinen Amiga-Programmen (unter Intuition) sollten Sie – nur in Lattice C – *_main()* anstatt *main()* einsetzen. Der Linker wird dann einen kürzeren Startup-Code einbinden, der nicht mehr die Standard-I/O-Kanäle öffnet. Damit sind zwar Funktionen wie *printf()* nicht mehr verwendbar, aber die brauchen Sie ja nicht unter Intuition.

Amiga.lib zuerst

Wenn Sie *printf()* und Konsorten einsetzen und damit nur ganze Zahlen, Zeichen und Strings ausgeben wollen, brauchen Sie die Standard-Library Ihres Compilers nicht, denn diese Funktionen stehen schon im ROM des Amiga. Dazu linken Sie »amiga.lib« vor Ihre C-Lib (siehe 3.2, 3.3), und schon wird der ganze printf()-Code nicht mehr in Ihr Programm eingebunden.

Das ROM und die »Protos« nutzen

Wieso soll man eigentlich die aufwendigen Libraries der Unix-Welt einbinden, wo doch der Amiga analoge Funktionen schon im ROM hat? Ein *printf("Hello World")*, in einem File namens *hallo.c* einfach so mit »lc –L hallo« kompiliert und gelinkt, erzeugt einen Programm-Code von 6372 Byte (Lattice C, Version 5.0). Das gleiche Ergebnis erreiche ich jedoch auch mit 196 Byte, wohl gesagt, immer noch in C. Nur in Assembler geht das noch kürzer (144 Byte). Listing 3.5.5 zeigt die Lösung.

```
#include "libraries/dos.h"
#include "proto/exec.h"
#include "proto/dos.h"

#define ABSEXECBASE ((struct ExecBase **)4L)
struct DosLibrary  *DOSBase;

void hallo()
{
 if (!(DOSBase = (struct DosLibrary *)
                 OpenLibrary("dos.library",0)))
      return;
 Write(Output(),"Hello World\n",12);
 CloseLibrary((struct Library *)DOSBase);
}
```

Listing 3.5.5: Der Kürze-Rekord: nur 196 Byte Code.

Typisch für Lattice sind die Includes aus dem Proto-Directory. Damit wird ein Dilemma von C ausgeschaltet, nämlich: C bringt alle Parameter einer Funktion auf den Stack, die Systemroutinen erwarten sie jedoch in Registern. Folglich muß der Compiler (Linker) Code einbauen, der die Parameter vom Stack holt und in Register umlädt. Genau das wird vermieden, wenn man die Proto-Include-Files anzieht. Der Aufruf der Systemroutinen sieht dann allerdings auch nicht mehr sehr nach C aus, sondern eher wie Assembler-Makros. Wenn Sie den Text als *hallo.c* speichern, können Sie das Programm kompilieren und linken mit

```
lc -v -y hallo.c
blink hallo.o
```

Maßgebend ist dabei, daß die Ausgabe über die Write()-Funktion des Amiga-DOS läuft. Weshalb soll ich da also mit »Amiga.Lib« linken? Nach dem Kompilieren mit der v-Option des Lattice C (kein Code zum Stack-Checking), die hier obligatorisch ist, reicht deshalb auch ein schlichtes *blink hallo.o*. Der Linker fügt jetzt nur den Header (16 Byte) hinzu, damit der Amiga »hallo« als ausführbares Programm anerkennt.

3.5.6 Compiler/Linker-Aufruf im Quelltext

Den folgenden Trick habe ich zuerst auf einer Public-Domain-Disk gesehen und bestaunt. Nach dem dritten Hingucken hatte ich es dann kapiert. Der Trick stammt von John Meissen. Schreiben Sie ein Programm – diesmal für Aztec C – doch einmal so:

```
echo; /*
cc test.c
ln test -lc
 quit
*/
main()
{
  printf("echo hat den Wert %d\n", echo);
}
```

Wenn Sie *lc –L test.c* einsetzen, geht das in Lattice C. Das Programm wird jetzt kompiliert und gelinkt (Name *test.c*), indem man aufruft

```
execute test.c
```

Des Rätsels Lösung: Das Execute des CLI gibt nach »echo« eine Leerzeile aus, den Rest der Zeile ignoriert es wegen des Semikolons, weil das im CLI einen Kommentar einleitet. Danach führt *execute* die Compile- und Link-Anweisungen aus, mehr nicht, denn jetzt folgt *quit*. Der C-Compiler sieht *echo;* und – das ist der Trick – legt deshalb die int-Variable *echo* an. Hier gilt nämlich die Regel, daß in C alles vom Typ *int* ist, sofern nicht anderes gesagt wird. Für den Compiler sind alle Zeilen nach *echo;* wegen des »/*...*/« aber Kommentare. Sie können *echo* durchaus auch als C-Variable benutzen. Nur um zu zeigen, daß es geht, habe ich ihr den Wert 4711 zugewiesen.

3.5.7 Trickreiche Makros

Wir haben ja schon öfter Makros kennengelernt, aber die waren alle relativ klein. Doch eine Größenbegrenzung gibt es nicht. Sie können mit dem Zeilenfortsetzungszeichen (\) immer weiter schreiben. Sie sollten dann nur den ganzen Ausdruck in Klammern setzen. Sie können zum Beispiel durchaus dieses Makro schreiben

```
#define OPENWINDOW(t)   ( Window = (struct Window *)
\
            open_window(0,0,640,256,(t), \
            WINDOWCLOSE | WINDOWSIZING | ACTIVATE,
\
            CLOSEWINDOW | VANILLAKEY,\
            NULL) )
```

Dann können Sie es aufrufen mit zum Beispiel

```
OPENWINDOW("Fenstertitel");
```

Das spart natürlich noch nichts. Doch wenn Sie dieses Makro und andere in ein Header-File packen, brauchen Sie dieses nur noch zu »includen«, und schon stehen alle Makros zur Verfügung. Durchforsten Sie einmal Ihre Programme auf immer wiederkehrende »Bandwürmer«, und Sie werden viele Ansätze für Ihr Makro-H-File finden. Mehr zum Thema finden Sie im Kapitel 4.4.1 im Zusammenhang mit dem Makro *zufallszahl*.

3.6 Zusatz-Hardware und andere Empfehlungen

Es gibt einige Möglichkeiten, sich das Programmierer-Leben etwas leichter zu machen, wenn man in Hardware investiert. Zur Auswahl haben Sie allerdings nur drei Dinge, nämlich eine Festplatte, viel RAM oder einen schnelleren Rechner. Lassen wir den Amiga 3000 einmal außen vor, so bleibt die Frage »Platte oder RAM?«.

Eine Festplatte macht Tempo

Zuerst etwas zur Zeitfrage: Für das komplette Malprogramm benötige ich mit einem Festplattensystem im »langsamen« Filessystem unter Aztec C 44 Sekunden zum Kompilieren und Linken, unter Lattice C 63 Sekunden. Der Unterschied zwischen den beiden Compilern ist also nicht sehr groß, er ist aber doch ziemlich riesig im Vergleich zu einem Disketten-System. Daraus sollten Sie folgern, daß eine Festplatte immer lohnt.

RAM ist noch schneller, oder?

Man kann die o.g. Zeiten etwa auf ein Zehntel bringen, wenn man das ganze System im RAM hält. Dazu gehören nicht nur der Compiler und der Linker, sondern auch alle H-Files und die Libraries sowie diverse CLI-Kommandos. So ab 2 Mbyte kann man schon allerhand machen, 4 Mbyte sind besser. Haben Sie jedoch keine Festplatte, dauert es nahezu ewig, bis das alles von den Disketten in das RAM geladen ist. Nun sagen Sie nicht, das ist eine Einmal-Zeit. Bei so manchem Absturz – und dafür gibt es laut Kapitel 3.1 sehr viele Gründe – ist nämlich die Prozedur wieder fällig. Deshalb ist man auch gezwungen, immer den letzten Stand der Quell-Files erst auf die Disk zu sichern, bevor man das Programm testet. Das ist natürlich ein extra Schritt, der bei der Festplattenlösung entfällt.

Daraus folgt: Entscheiden Sie sich zuerst für eine Festplatte. Nur Disketten und viel RAM sind nicht zu empfehlen, viel RAM plus eine Festplatte ist die Ideallösung.

Vorkompilierte Header-Files einsetzen

Auch die Header(Include)-Files müssen natürlich vom Compiler übersetzt werden, und das bei jedem Lauf neu. Man kann einiges sparen, wenn man diese Files einmal übersetzt und dann dem Compiler sagt, daß er bitte diese Übersetzung benutzen soll. Hier das Kochrezepte:

Schreiben Sie einen Quelltext, in dem alle Includes stehen, die Sie brauchen, zum Beispiel diese:

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfx.h>
#include <graphics/text.h>
```

Speichern Sie die Datei unter dem Namen *x.c* und verfahren jetzt je nach Compiler weiter.

Für Aztec C:

Erzeugen Sie das sogenannte Dump-File mit

```
cc -ho x.dmp x.c
```

Ist das erledigt, können Sie Ihre Programme – zum Beispiel *prog.c* so übersetzen.

```
cc -hi x.dmp prog.c
```

Für Lattice C:

Erzeugen Sie das sogenannte Dump-File mit

```
lc -ph -ox.dmp x
```

Ist das erledigt, können Sie Ihre Programme – zum Beispiel *prog.c* so übersetzen.

```
lc -L -Hx.dmp prog
```

Für Aztec und Lattice:

Für beide Compiler gilt: Die Quelltexte bleiben völlig unverändert. Auch die Includes müssen in den Quelltexten bleiben. Sie können deshalb auch wahlweise mit und ohne dieses dmp-File kompilieren.

☞ Die dmp-Files müssen mit demselben Speichermodell kompiliert werden wie das Hauptprogramm. Optionen wie *-mc* (Aztec) oder *-b0* (Lattice) sind also in beiden Fällen anzuwenden.

Kapitel 4: Referenz

In diesem Kapitel finden Sie die Referenzen zu den Themenkreisen des Buchs, aber auch einiges darüber hinaus. Wie zum Beispiel den Abschnitt »Schnellinformation«, der eine Reihe von Standardlösungen zu häufigen Programmierproblemen der Praxis beinhaltet.

4.1 Die wichtigsten Standard-C-Funktionen

Im folgenden finden Sie eine kleine Auswahl von ANSI-C-Funktionen. Darüber hinaus bieten Aztec und Lattice spezielle Funktionen für jeweils nur diesen Compiler und spezielle Amiga-Funktionen, die sich gleichfalls unterscheiden. Sie sollten auf jeden Fall das »Library-Manual« einmal durchgehen, um all die vorhandenen Möglichkeiten Ihres Compilers zu kennen.

abs(i)	Absolutwert einer vorzeichenbehafteten int-Zahl. *labs()* für *long*, *fabs()* für *doubble*.
atof(s)	Konvertiert String in double-Zahl
atoi(s)	Konvertiert String in int-Zahl
cos(d)	Ergibt Cosinus einer double-Zahl
div(i)	int-Ganzzahl-Division
exit()	Verläßt das Programm und gibt malloc()-Speicher frei
getc()	Liest ein Zeichen von der Tastatur
gets()	Liest eine Zeile in einen String
is...(c)	Die folgenden Funktionen aus *ctype.h* ergeben TRUE (nicht 0), wenn die Bedingung erfüllt ist.

isalpha(c)	c ist ein Buchstabe
isupper(c)	c ist ein Großbuchstabe
islower(c)	c ist ein Kleinbuchstabe
isdigit(c)	c ist eine Ziffer
isalnum(c)	c ist alphanumerisch
isspace(c)	c ist ein Blank, Tab, CR, LF oder FF
ispunct(c)	c ist ein Punktuationszeichen
isprint(c)	c ist ein druckbares Zeichen (Code 32 bis 127)
log(d)	Natürlicher Logarithmus, sonst *log10(d)*
pow(x,y)	Ergibt x hoch y (in *double*)
putc(c)	Ausgabe eines Zeichens
puts(s)	Ausgabe eines Strings
remove("name")	Löscht die Datei »*name*«
sqrt(d)	Quadratwurzel einer double-Zahl
strcat(z,s)	Hängt String *s* an String *z* an
strncat(z,s,n)	Hängt String *s* an String *z* an, Länge aber max. n
strcmp(s1,s2)	Vergleicht beide Strings, 0 wenn gleich
strspn(s1,s2)	Ergibt Beginn von s1 in s2, wenn enthalten
tan(d)	Tangens einer double-Zahl
tolower(c)	Konvertiert Buchstaben in Kleinbuchstaben
toupper(c)	Konvertiert Buchstaben in Großbuchstaben

4.2 Die wichtigsten Amiga-Funktionen

Der Amiga bietet in seinen Libraries mehrere hundert Funktionen, von denen ich hier nur die meiner Meinung nach wichtigsten aufführen kann. Darüber kann man streiten, nicht jedoch über folgenden Tip: Bevor Sie das berühmte Rad neu erfinden, informieren Sie sich. Im Anhang A3 gibt es einige Literaturempfehlungen.

4.2.1 Die wichtigsten Funktionen von Intuition

AddGadget(Window, Gadget, Position)

Fügt ein Gadget in die Gadget-Liste ein.

Window	Zeiger auf das Window
Gadget	Zeiger auf das neue Gadget
Position	Platz in der Liste

☞ Am sichersten ist Position = −1, womit das Gadget an das Ende der Liste gehängt wird.

AllocRemember(RememberKey, Size, Flags)

Reserviert einen Speicherblock durch Aufruf der Exec-Funktion *AllocMem()* und legt einen Knoten in der Intuition-Liste zur Verwaltung dieser Speicherblöcke an.

RememberKey	Zeiger auf Knoten (setzen Sie den zuerst auf NULL)
Size	Größe in Bytes
Flags	Speichertyp (Fast-Mem, Chip-Mem)

AutoRequest(Window, BodyText, PosText, NegText, PFlag, NFlag, W, H)

Baut automatisch einen Requester auf und wartet auf Antwort.

Window	Zeiger auf eine Window-Struktur, wenn NULL, wird ein Window angelegt.
BodyText	Zeiger auf IntuiText-Struktur mit der Frage
PosText	Zeiger auf IntuiText-Struktur mit positiver Antwort (Ja)
NegText	Zeiger auf IntuiText-Struktur mit negativer Antwort (Nein)
PFlag	IDCMP-Flags, die anstatt positiver Antwort wirken
NFlag	DCMP-Flags, die anstatt negativer Antwort wirken
W	Weite des Requesters
H	Höhe

BuildSysRequest(Window, BodyText, PosText, NegText, PFlag, NFlag, W, H)

Prinzipiell wie *AutoRequest()*, nur mit einem Unterschied: Für die Flags müssen Sie die IDCMP-Flags einsetzen, mit denen das Window, das diese Funktion erzeugt, initialisiert wird. Die Funktion gibt einen Zeiger auf diese Window-Struktur zurück. Dann müssen Sie auch mit *Wait()/GetMsg()* usw. den IDCMP-Port wie üblich bedienen.

Siehe auch *FreeSysRequest()*.

ClearDMRequest(Window)

Entfernt einen DM-Requester aus einem Window.

Siehe auch SetDMRequest().

ClearMenuStrip(Window)

Löscht den Menu-Strip eines Windows.

Siehe auch *CloseWindow()* und *SetMenuStrip()*.

ClearPointer (Window)

Entfernt einen selbstdefinierten Mauszeiger und ersetzt ihn durch den Standard-Mauszeiger.

CloseWindow(Window)

Schließt ein Window und gibt den Speicher wieder frei. Eventuell noch unbeantwortete Messages sind verloren. Vergessen Sie nie, einen *Menu-Strip* zu löschen, sonst bleibt der im Screen stehen. Drückt man dann die Menü-Taste (linke Maustaste), stürzt das System ab.

DisplayAlert (AlertNumber, String, Height)

Erzeugt einen System-Alarm, siehe Sitzung 8.

DisplayBeep (Screen)

Läßt den Screen einmal kurz aufblitzen. Im Falle eines NULL-Arguments blitzen alle Screens (oder der einzig vorhandene).

EndRequest (requester, Window)

Entfernt den Requester aus einem Window.

requester	Zeiger auf Requester-Struktur
Window	Zeiger auf Window-Struktur

FreeRemember (RememberKey, ReallyForget)

Gibt den mit *AllocRemember()* reservierten Speicher wieder frei.

RememberKey	Zeiger auf RememberKey-Struktur
ReallyForget	Nur TRUE sinnvoll

FreeSysRequest (Window)

Entfernt den mit *BuildSysRequest()* gebauten Requester. Sie dürfen diese Funktion keinesfalls aufrufen, wenn *BuildSysRequest()* keinen Zeiger (NULL) auf eine Window-Struktur zurückgegeben hatte.

GetDefPrefs(PrefBuffer, Size)

Preferenz-Struktur oder Teile davon lesen. Wenn Intuition das erstemal aufgerufen wird, speichert es die Werte auf der Disk als Defaults ab. Die aktuellen (veränderten) Daten werden woanders abgelegt. Mit dieser Funktion lesen Sie die Defaults.

Prefbuffer	Zeiger auf einen Puffer, den Sie stellen müssen, und der mindestens *Size* Bytes groß sein muß.
Size	Anzahl Bytes, die eingelesen werden sollen.

Es ist nicht nötig, die ganze Preference-Struktur zu lesen. Die wichtigsten Daten stehen vorne.

GetPrefs(PrefBuffer, Size)

Wie vor, nun aber für die aktuellen Werte.

IntuiTextLength(itext)

Gibt die Länge eines Textes zurück (in Pixel), auf den das Textfeld der IntuiText-Struktur *itext* zeigt.

ModifyIDCMP(Window, Neue_Flags)

Überschreibt die IDCMP-Flags des Windows mit den neuen Werten. Hatte das Window bis dahin keine Flags, werden die IDCMP-Ports neu angelegt.

MoveWindow(Window, dx,dy)

Bewegt das Window um dx,dy Pixel. Die Funktion gibt kein Ergebnis zurück und testet nicht auf Fehler. Bei falschen Werten kann das Programm abstürzen.

OffGadget(Gadget, Window, Req)

Das Gadget wird »disabled« und grau gezeichnet (kann nicht bedient werden.

Gadget	Zeiger auf die Gadget-Struktur
Window	Zeiger auf die Window-Struktur
Req	Zeiger auf den Requester, falls das Gadget Teil eines solchen ist, sonst NULL.

OffMenu(Window, MenuNumber)

»Disabled« (kann nicht bedient werden, wird grau gezeichnet) ein Menü, ein Item oder Sub-Item und alles, was diesen folgt.

OnGadget(Gadget, Ptr, Req)

Die Umkehr der Funktion *OffGadget()*, d.h., das Gadget ist wieder wählbar. Parameter siehe *OffGadget()*.

OnMenu(Window, MenuNumber)

Die Umkehr der Funktion *OffMenu()*. Parameter siehe *OffMenu()*.

OpenWindow(NewWindow)

Öffnet ein neues Window. *NewWindow* ist ein Zeiger auf die zuvor angelegte NewWindow-Struktur.

PrintIText(rp, itext, x,y)

Druckt einen Text, der mit all seinen Merkmalen in der Struktur vom Typ *IntuiText* steht, ab der Position *x,y*. *rp* ist ein Zeiger auf den RastPort.

RefreshGadgets(Gadget, Window, Req)

Zeichnet alle Gadgets ab *Gadget* neu. Das ist erforderlich, wenn Sie Gadget-Parameter geändert haben, *AddGadget()* oder *RemoveGadget()* aufgerufen wurden oder der Verdacht besteht, daß eine Grafik-Routine die Gadgets übermalt hat.

Gadget	Zeiger auf die Gadget-Struktur
Window	Zeiger auf die Window-Struktur
Req	Zeiger auf den Requester, falls das Gadget Teil eines solchen ist, sonst NULL.

RemoveGadget(Window, Gadget)

Entfernt ein Gadget aus der Liste.

ReportMouse(Window, Boolean)

Damit wird das ständige Melden von Maus-Events (das viel Zeit kostet) an- (Boolean = TRUE) oder ausgeschaltet (Boolean = 0).;

Request(Requester, Window)

Damit wird ein Requester gezeichnet.

Requester	Zeiger auf die Requester-Struktur
Window	Zeiger auf die Window-Struktur

SetDMRequest(Window, req)

Erlaubt, daß ein DM-Requester erscheinen kann, *req* ist ein Zeiger auf eine Requester-Struktur.

SetMenuStrip(Window, Menu)

Schaltet das Menü ein, auf dessen Struktur *Menu* zeigt.

SetPointer(Window, Pointer, Height,Width,Xoffset,Yoffset)

Schaltet einen neuen Mauszeiger ein.

Siehe Kapitel 3.5.

SetPrefs(PrefBuffer, Size, flag)

Schreibt die (geänderten) Daten in *PrefBuffer* zurück und läßt die neuen Einstellungen wirksam werden.

Prefbuffer	Zeiger auf den Puffer mit den Daten
Size	Anzahl Bytes, die geschrieben werden sollen.
flag	TRUE: Daten gehen auf Disk und in RAM FALSE: Daten gehen nur in RAM

SizeWindow(Window, dx,dy)

Ändert die Größe eines Windows um die dx/dy-Beträge. Es findet keine Kontrolle statt. Falsche Werte können zum Absturz führen.

WindowLimits(Window, minwidth,minheight, maxwidth,maxheight)

Setzt die entsprechenden Parameter, die sonst in der NewWindow-Struktur vorgegeben werden, neu.

WindowToBack(Window)
WindowToFront(Window)

Bringt ein Window in den Hinter- bzw. Vordergrund.

4.2.2 Die wichtigsten Grafik-Funktionen

In den folgenden Grafik-Funktionen ist *rp* immer der Zeiger auf den RastPort.

Draw(rp, x,y)

Zeichnet eine Linie vom aktuellen Cursor-Punkt zum Punkt *x,y*.

DrawCircle(rp, x,y, radius)

Zeichnet einen Kreis mit dem Radius *radius* um den Punkt x,y.

DrawEllipse(rp, x,y, h_radius, v_radius)

Zeichnet eine Ellipse mit dem horziontalen Radius *h_radius* und dem vertikalen Radius *v_radius* um den Punkt x,y.

Flood(rp, modus, x,y)

Füllt einen Bereich ab dem Punkt *x,y*. Ist *modus==1*, gilt das so lange, bis auf die Außenlinien getroffen wird. Ist *modus==0*, wird so lange gefüllt, bis Punkte der mit *SetOpen()* gesetzten Farbe angetroffen werden.

Move(rp, x,y)

Setzt den Grafik-Cursor auf den Punkt *x,y*.

RectFill(rp, links,oben, rechts,unten)

Zeichnet ein gefülltes Rechteck in der aktuellen SetAPen()-Farbe.

SetAPen(rp, Farbregisternummer)

Setzt die Zeichenfarbe. In der Normalauflösung sind die Register 0 bis 3 möglich.

SetBPen(rp, Farbregisternummer)

Setzt die Hintergrundfarbe für Funktionen, die sie nutzen. In der Normalauflösung sind die Register 0 bis 3 möglich.

SetDrPt(rp, Musterwort)

Setzt das Muster, mit dem Linien gezeichnet werden, Beispiel: *SetDrPt(rp, 0xAAAA);*.

Text(rp, string, anzahl)

Gibt vom Text *string anzahl* Zeichen aus.

WritePixel(rp, x,y)

Setzt einen einzelnen Punkt an der Position *x,y*.

4.2.3 Die wichtigsten DOS-Funktionen

In den folgenden Funktionen ist *fh* immer das File-Handle, das die Open()-Funktion zu einem File liefert. Am einfachsten deklarieren Sie *fh* als eine Variable vom Typ ULONG. Außerdem ist *name* ein String oder eine Stringvariable.

Close(fh)

Schließt eine zuvor mit *Open()* geöffnete Datei. Schließen Sie nie eine Datei, die Sie nicht selbst geöffnet haben.

DeleteFile(name)

Löscht die Datei *name*. Die Funktion gibt TRUE zurück, wenn die Aktion Erfolg hatte, sonst FALSE.

Input()

Liefert das Eingabe-File-Handle des CLI-Windows, in dem das Programm gestartet wurde.

IoErr()

Liefert die Nummer des zuletzt aufgetretenen DOS-Fehlers.

Open(name, modus)

Öffnet die Datei *name* und liefert ein File-Handle *(fh)* oder 0, wenn die Datei nicht geöffnet werden konnte. Mit dem *modus* MODE_OLDFILE wird eine existierende Datei geöffnet, mit MODE_NEWFILE eine neue angelegt.

Output()

Liefert das Ausgabe-File-Handle des CLI-Windows, in dem das Programm gestartet wurde.

Ist_Zahl = Write(fh, Puffer, Soll_Zahl);
Ist_Zahl = Read (fh, Puffer, Soll_Zahl);

Es werden *Soll_Zahl* Bytes aus dem Puffer auf die Disk geschrieben, bzw. in ihn eingelesen. Der Puffer ist eine Variable, die Sie vor dem Schreiben mit Daten füllen müssen, beim Lesen werden die Daten von der Diskette in den Puffer kopiert. Nach den Operationen steht in *Ist_Zahl*, wie viele Bytes tatsächlich geschrieben bzw. gelesen worden sind. *Ist_Zahl* ist besonders beim Lesen von Bedeutung. Da es in Amiga-DOS keine EOF-Funktion (End Of File) gibt, müssen Sie so lange lesen, bis *Ist_Zahl* null ist.

Rename(alt_name, neu_name)

Die Datei wird umbenannt. Die Funktion gibt TRUE zurück, wenn die Aktion Erfolg hatte, sonst FALSE.

Seek(fh, position, mode)

Stellt den File-Zeiger (die Position, ab der gelesen bzw. geschrieben wird, auf *position*. Dabei wird in Abhängigkeit von *mode* so gezählt:

OFFSET_BEGINNING	Ab Beginn
OFFSET_CURRENT	Ab aktueller Position
OFFSET_END	Ab Datei-Ende rückwärts gezählt

Die Funktion gibt die Position des File-Zeigers vor der Aktion zurück.

4.3 Die wichtigsten Amiga-Datenstrukturen

Auch hier gibt es nur eine kleine Auswahl und ansonsten den Hinweis auf die Literaturtips im Anhang A3.

Die Datenstrukturen von Intuition

```
struct NewScreen
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT Depth;
    UBYTE DetailPen, BlockPen;
    USHORT ViewModes;
    USHORT Type;
    struct TextAttr *Font;
    UBYTE *DefaultTitle;
    struct Gadget *Gadgets;
    struct BitMap *CustomBitMap;
};
struct Screen
{
    struct Screen *NextScreen;
    struct Window *FirstWindow;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT MouseY, MouseX;
    USHORT Flags;
    UBYTE *Title;
    UBYTE *DefaultTitle;
    struct TextAttr *Font;
    struct ViewPort ViewPort;
    struct RastPort RastPort;
    struct BitMap BitMap;
    struct Layer_Info LayerInfo;
    struct Gadget *FirstGadget;
    UBYTE DetailPen, BlockPen;
    USHORT SaveColor0;
    struct Layer *BarLayer;
    UBYTE *ExtData;
    UBYTE *UserData;
};
```

```
 struct NewWindow
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    UBYTE DetailPen, BlockPen;
    ULONG IDCMPFlags;
    ULONG Flags;
    struct Gadget *FirstGadget;
    struct Image *CheckMark;
    UBYTE *Title;
    struct Screen *Screen;
    struct BitMap *BitMap;
    SHORT MinWidth, MinHeight;
    USHORT MaxWidth, MaxHeight;
    USHORT Type;
};

struct Window
{
    struct Window *NextWindow;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT MouseY, MouseX;
    SHORT MinWidth, MinHeight;
    USHORT MaxWidth, MaxHeight;
    ULONG Flags;
    struct Menu *MenuStrip;
    UBYTE *Title;
    struct Requester *FirstRequest;
    struct Requester *DMRequest;
    SHORT ReqCount;
    struct Screen *WScreen;
    struct RastPort *RPort;
    BYTE BorderLeft, BorderTop;
    BYTE BorderRight, BorderBottom;
    struct RastPort *BorderRPort;
    struct Gadget *FirstGadget;
    struct Window *Parent, *Descendant;
    USHORT *Pointer;
    BYTE PtrHeight;
    BYTE PtrWidth;
    BYTE XOffset, YOffset;
```

```
    ULONG IDCMPFlags;
    struct MsgPort *UserPort, *WindowPort;
    struct IntuiMessage *MessageKey;
    UBYTE DetailPen, BlockPen;
    struct Image *CheckMark;
    UBYTE *ScreenTitle;
    SHORT GZZMouseX;
    SHORT GZZMouseY;
    SHORT GZZWidth;
    SHORT GZZHeight;
    UBYTE *ExtData;
    BYTE *UserData;
    struct Layer *WLayer;
    struct TextFont *IFont;
};

struct Menu
{
    struct Menu *NextMenu;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    BYTE *MenuName;
    struct MenuItem *FirstItem;
    SHORT JazzX, JazzY, BeatX, BeatY;
};

struct MenuItem
{
    struct MenuItem *NextItem;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    LONG MutualExclude;
    APTR ItemFill;
    APTR SelectFill;
    BYTE Command;
    struct MenuItem *SubItem;
    USHORT NextSelect;
};

struct Gadget
```

```
{
    struct Gadget *NextGadget;
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    USHORT Flags;
    USHORT Activation;
    USHORT GadgetType;
    APTR GadgetRender;
    APTR SelectRender;
    struct IntuiText *GadgetText;
    LONG MutualExclude;
    APTR SpecialInfo;
    USHORT GadgetID;
    APTR UserData;
};

struct Requester
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height;
    SHORT RelLeft, RelTop;
    struct Gadget *ReqGadget;
    struct Border *ReqBorder;
    struct IntuiText *ReqText;
    USHORT Flags;
    UBYTE BackFill;
    struct Layer *ReqLayer;
    UBYTE ReqPad1[32];
    struct BitMap *ImageBMap;
    struct Window *RWindow;
    UBYTE ReqPad2[36];
};

struct BoolInfo
{
    USHORT Flags;
    UWORD  *Mask;
    ULONG  Reserved;
};

struct PropInfo
{
```

```
    USHORT Flags;
    USHORT HorizPot;
    USHORT VertPot;
    USHORT HorizBody;
    USHORT VertBody;      USHORT CWidth;
    USHORT CHeight;
    USHORT HPotRes, VPotRes;
    USHORT LeftBorder;
    USHORT TopBorder;
};

struct StringInfo
{
    UBYTE *Buffer;
    UBYTE *UndoBuffer;
    SHORT BufferPos;
    SHORT MaxChars;
    SHORT DispPos;
    SHORT UndoPos;
    SHORT NumChars;
    SHORT DispCount;
    SHORT CLeft, CTop;
    struct Layer *LayerPtr;
    LONG LongInt;
    struct KeyMap *AltKeyMap;
};

struct IntuiText
{
    UBYTE FrontPen, BackPen;
    UBYTE DrawMode;
    SHORT LeftEdge;
    SHORT TopEdge;
    struct TextAttr *ITextFont;
    UBYTE *IText;
    struct IntuiText *NextText;
};

struct Border
{
    SHORT LeftEdge, TopEdge;
    UBYTE FrontPen, BackPen;
```

```
    UBYTE DrawMode;
    BYTE Count;
    SHORT *XY;
    struct Border *NextBorder;
};

 struct Image
{
    SHORT LeftEdge;
    SHORT TopEdge;
    SHORT Width;
    SHORT Height, Depth;
    USHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
};

struct IntuiMessage
{
    struct Message ExecMessage;
    ULONG Class;
    USHORT Code;
    USHORT Qualifier;
    APTR IAddress;
    SHORT MouseX, MouseY;
    ULONG Seconds, Micros;
    struct Window *IDCMPWindow;
    struct IntuiMessage *SpecialLink;
};

struct Remember
{
    struct Remember *NextRemember;
    ULONG RememberSize;
    UBYTE *Memory;
};

struct Preferences
{
    BYTE FontHeight;
    UBYTE PrinterPort;
    USHORT BaudRate;
```

```
struct timeval KeyRptSpeed;
struct timeval KeyRptDelay;
struct timeval DoubleClick;
USHORT PointerMatrix[POINTERSIZE];
BYTE XOffset;
BYTE YOffset;
USHORT color17;
USHORT color18;
USHORT color19;
USHORT PointerTicks;
USHORT color0;
USHORT color1;
USHORT color2;
USHORT color3;
BYTE ViewXOffset;
BYTE ViewYOffset;
WORD ViewInitX, ViewInitY;
BOOL EnableCLI;
USHORT PrinterType;
UBYTE PrinterFilename[FILENAME_SIZE];
USHORT PrintPitch;
USHORT PrintQuality;
USHORT PrintSpacing;
UWORD PrintLeftMargin;
UWORD PrintRightMargin;
USHORT PrintImage;
USHORT PrintAspect;
USHORT PrintShade;
WORD PrintThreshold;
USHORT PaperSize;
UWORD PaperLength;
USHORT PaperType;
UBYTE   SerRWBits;
UBYTE   SerStopBuf;
UBYTE   SerParShk;
UBYTE   LaceWB;
UBYTE   WorkName[FILENAME_SIZE];
BYTE    sys_reserved1;
BYTE    sys_reserved2;
UWORD       PrintFlags;
UWORD       PrintMaxWidth;
UWORD       PrintMaxHeight;
```

```
    UBYTE       PrintDensity;
    UBYTE       PrintXOffset;
    UWORD       wb_Width;
    UWORD       wb_Height;
    UBYTE       wb_Depth;
    UBYTE       ext_size;
    BYTE    ext_bytes[PREF_EXTBYTES];
};
```

4.4 Schnellinformation: Einige Standard-Lösungen in C

Die folgenden Programme und ihre Beschreibungen sollen Ihnen einen kleinen Einblick in die vielfältigen Möglichkeiten der Sprache C vermitteln. Merke: Es gibt nichts, was in C nicht lösbar ist, und kaum etwas, was nicht schon gelöst wurde.

Sortieren und Zufallszahlen

Ein Standardproblem in der Datenverarbeitung ist das Sortieren von Arrays, wobei diese nicht nur Zahlen, sondern auch Texte enthalten können. Hierfür stellt C mit der qsort()-Funktion einen der besten Algorithmen, nämlich den Quick-Sort, zur Verfügung. Die Funktion setzt voraus, daß Sie eine sogenannte Vergleichsfunktion zur Verfügung stellen. Daß diese recht einfachen Funktionen nicht »eingebaut« sind, ist ein Vorteil. Damit werden nämlich die Sortierroutinen unabhängig vom Datentyp. Bei der Gelegenheit – mit irgend etwas müssen wir den Array ja füllen – bauen wir noch ein praktisches Makro für Zufallszahlen.

Das Programm laut Listing 4.4.1 soll

- einen Array mit 1000 Zufallszahlen füllen,
- jede zehnte davon ausgeben,
- den Array sortieren
- und ihn wieder ausgeben.

```
/* SORT.C
   Demo zu Sortieren, Suchen und Zufallszahlen
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

#define  ANZAHL 1000

int array[ANZAHL];

/*
  Makro zur Erzeugung einer Zufallszahl
  im Bereich min...max
  ------------------------------------ */
#define zufallszahl( min, max ) ((rand() % \
        (int)( ( (max) + 1) - (min) ) ) + (min) )

/* Prototypen
   ---------- */
void PrintArray(void);
int cmp ();

void main()
{
  int i;

  /* Setze Startwert des Zufallsgenerators
     auf die Systemzeit
     ---------------------------------- */
  srand( (unsigned) time( NULL ) );

  /* Fülle Array mit Zufallszahlen
     ---------------------------- */
  printf( "Array wird geladen...\n" );
```

Listing 4.4.1: (Fortsetzung folgende Seiten)

```
  for( i = 0; i < ANZAHL; i++ )
    array[i] = zufallszahl( 1, 999 );

  /* Weil der Array so groß ist, zeige nur

     jedes zehnte Element
     ------------------------------------ */
  PrintArray();

  /* Sortieren nach dem Quicksort-Algorithmus
     --------------------------------------- */
  printf( "\nIch Sortiere...\n" );
  qsort(
      (void *)array , /* Ab wo sortieren    */
      ANZAHL,         /* Anzahl Elemente    */
      sizeof( int ),  /* Bytes pro Element  */
      cmp );          /* Vergleichsfunktion */

  PrintArray();
  puts("");

}

/* Zeige jedes zehnte Element des sortierten Arrays
   ------------------------------------------------ */
void PrintArray(void)
{
  int i, j;
  printf("\nJedes zehnte Element des Arrays:\n");
  for( i = 9, j=1; i < ANZAHL; i += 10,j++ )
    {
      printf("%4d", array[i] );
      if( !(j % 18) )
         printf( "\n" );
    }
}
/*  Für qsort()  muß eine Funktion bereitgestellt
```

Listing 4.4.1: (Fortsetzung folgende Seite)

```
    werden, die bei Gleichheit 0 zurück gibt,
    bei elem1 > elem2 1  und umgekehrt -1.
   ------------------------------------------ */
int cmp( int *elem1, int *elem2 )
{
  if( *elem1 > *elem2 )
    return 1;
  else if( *elem1 < *elem2 )
    return -1;
  else
    return 0;
}
```

Listing 4.4.1: Suchen und Sortieren in C. So ein Zufall?

Das Makro *Zufallszahl* liefert eine Zufallszahl im Bereich von *min* bis *max*. Der Trick dabei: Die Funktion *rand()* liefert eine Zufallszahl im Bereich von null bis MAX_RAND, einer Konstanten, die meistens als 32767 definiert ist. Um diese Zahl auf *max* zu begrenzen, nimmt man den Modulo-Operator. Zum Beispiel ist 32767 % 99 == 97 oder 32767 % 33 == 31. Sie sehen also, wir sind schon dicht dran, der Rest der Riesenformel sorgt für die Korrektur bzw. den richtigen min-Wert.

Makros müssen entweder in einer Zeile stehen oder mit dem Zeilenfortsetzungszeichen (\) umbrochen werden. Wird der Makro-Ausdruck umbrochen (oder enthält er Leerstellen), muß er in Klammern gesetzt werden. Schaden können die Klammern nie. Werden Argumente weiterentwickelt, muß man sie in Klammern setzen, auch hier können Klammern nie schaden.

Die Zufallsreihe selbst ist bei jedem Lauf dieselbe, es sei denn, man ändert mit *srand()* die »seed« (den Startwert). Damit der Startwert selbst wenigstens einigermaßen zufällig ist, setzt man hierfür üblicherweise die Systemzeit ein.

Den nun folgenden Funktionsaufruf für das Sortieren können Sie anhand der Kommentare sicherlich nachvollziehen. Das Beispiel ist auch deshalb so einfach, weil hier nur int-Zahlen zu bearbeiten sind. Wenn Sie einen anderen Typ bearbeiten wollen, ist das auch kein Problem. Im Falle von z.B. *double* müssen Sie nur beim Aufruf *sizeof(double)* schreiben, und in der Vergleichsfunktion auch *double* als Typ einsetzen. Manche Compiler bieten auch schon verschiedene *qsort*-Funktionen für die Standardtypen.

Etwas komplizierter wird die Sache bei Strings, wie Listing 4.4.2 zeigt.

```
/* TSORT.C
   Demo zum Sortieren von Texten
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define  ANZAHL 5

char *array[ANZAHL] = { "Meiers Franz",
            "Meiers Emil",
            "Fritze",
            "Anton",
            "Charly Brown"
              };

void PrintArray(void);
int cmp(), cmps();

void main()
{
  puts("Unsortiert:");
  PrintArray();

  /* Sortieren nach dem Quicksort-Algorithmus   */
  qsort( (void *)array,   /* Ab wo sortieren    */
         ANZAHL,          /* Anzahl Elemente    */
         sizeof(char *), /* Bytes pro Element  */
         cmps );          /* Vergleichsfunktion */

  puts("\nSortiert:");
  PrintArray();
}

void PrintArray(void)
```

Listing 4.4.2: (Fortsetzung folgende Seite)

```
{
  int i;
  for(i=0; i < ANZAHL; i++ )
    printf( "%s\n", array[i] );
}


int cmp( char **elem1,char **elem2 )
{
  if( **elem1 > **elem2 )
  return 1;
  else if( **elem1 < **elem2 )
  return -1;
  else
  return 0;
}

int cmps( char **elem1,char **elem2 )
{
  return strcmp(*elem1, *elem2);
}
```

Listing 4.4.2: So sortiert man Strings

Hier ist *array* ein Array von Strings, also auch ein *array* von Arrays des Typs *char*. Beachten Sie den kleinen Trick in der Deklaration. Man kann nämlich nicht *char array[ANZAHL][]* schreiben, sondern müßte dann zum Beispiel *array[ANZAHL][20]* tippen, was einen zwingt, die größte Stringlänge abzuzählen. Mit **array[ANZAHL]* hingegen kann man den Compiler überlisten. Der Zweck des Listings ist jedoch ein anderer. Die Vergleichsfunktion will Zeiger auf die Elemente sehen. Die String-Arrays sind aber schon Zeiger, also brauchen wir Zeiger auf Zeiger. Praktisch heißt das »noch ein Stern«, womit klar ist, warum es z.B. ***elem1* heißt.

☞ Und noch ein Hinweis: Hier heißt die Vergleichsfunktion *cmps()*. Diese Methode ist möglich, weil *strcmp()* genau die Werte –1, 0 bzw. 1 liefert, die von *qsort()* gefordert werden. Sie können aber genausogut *cmp()* einsetzen.

4.4.2 Das Mehrfachproblem und Lottozahlen

Sobald man einen Array sortiert hat, erkennt man sehr schön, ob einige Einträge mehrfach vorhanden sind, sie stehen dann ja nebeneinander. Wie man das programmtechnisch feststellt, zeigt Listing 4.4.3 auszugsweise.

```
#define  ANZAHL 6
int flag;

 do
  {
    flag=0;

          srand( (unsigned) time( NULL ) );
          for( i = 0; i < ANZAHL; i++ )
          array[i] = zufallszahl( 1, 49 );
          qsort( (void *)array,  ANZAHL,
                  sizeof( int ), cmp );

    for(i=0; i < ANZAHL-1; i++)
      if (array[i] == array[i+1] )
        flag = 1;
   if (flag)  printf(".");
  }while(flag);

  puts("");
  for(i=0; i<ANZAHL; i++)
    printf("%5d", array[i] );
  puts("");
```

Listing 4.4.3: Das Programm für die Lottozahlen (Auszug)

Die Zahlen selbst werden wie in Listing 4.4.1 erzeugt, nur daß hier ANZAHL auf 6 und der Zufallsbereich auf 1 bis 49 gesetzt wurden. Der Array wird sortiert, und nun haben wir das Problem. Von den sechs Zahlen können (müssen nicht) mehrere gleich sein. Unser Ansatz lautet: Wenn das der Fall ist, erzeuge sechs neue Zahlen, und das so lange, bis es sechs verschiedene sind. Reduziert man das Listing auf den Kern, sieht das so aus:

```
do
  {
    flag=0;
    /* lade Array */
    for(i=0; i < ANZAHL-1; i++)
      if (array[i] == array[i+1] )
        flag = 1;
  }while(flag);
```

Mit »if (array[i] == array[i+1]« wird verglichen, ob das aktuelle Element und sein Nachfolger gleich sind. Wenn das der Fall ist, wird *flag = 1* gesetzt, womit auch die do-Schleife wieder gestartet wird. Der Rest dient der Dekoration. So wird zum Beispiel für jeden Durchlauf ein Punkt ausgegeben, damit der Anwender sieht, daß sich etwas tut (100 Fehlversuche sind durchaus drin). Wichtiger ist folgendes: Die Schleife darf nur bis zum vorletzten Element laufen, hier *ANZAHL–1*, weil das letzte Element natürlich keinen Nachfolger hat.

4.4.3 File-I/O in Standard-C

In der zehnten Sitzung hatten wir uns mit Amiga-DOS befaßt und dabei festgestellt, daß damit geschriebene Programme nicht portabel sind, also nicht zum Beispiel auf einem PC laufen. Abhilfe bringt die Lösung in Standard-C. Hier unterscheidet man zwischen Textfiles und Binär-Files (allen anderen). Der Hauptunterschied liegt darin, daß in Textfiles die Zeichen mit den Codes 10 und 13 als LF (Zeilenvorschub) bzw. als CR (Return) interpretiert werden, und – jetzt kommt's – wie ein CR in ein CR + LF umgewandelt wird. Beginnen wir mit dem Textmodus laut Listing 4.4.4.

```
#include <stdio.h>

main()
{
 FILE *fp;
 char string[] = "Ein Beispiel-Text";

                              /* Öffne File      */
 if( (fp = fopen( "testfile","w" )) != NULL )
   {
      fputs( string, fp ); /* Schreibe String */
      fclose( fp );        /* Schließe File   */
```

```
    }
  else
    printf( "Fehler beim Schreiben\n");
                                /* Öffne File      */
   if( (fp = fopen( "testfile","r" )) != NULL )
    {
      fgets(string, 80, fp );  /* Lese String */
      puts( string );
      fclose( fp );            /* Schließe File   */
   }
   else
      printf( "Fehler beim Lesen\n");
}
```

Listing 4.4.4: Text-I/O in Standard-C: Das Prinzip

Ein File wird mit *fopen(DateiName, Modus)* geöffnet. Die Funktion gibt einen File-Zeiger zurück, alle weiteren Zugriffe laufen darüber. Ist der Return-Wert NULL, konnte das File nicht geöffnet werden. Der Filepointer wurde hier mit *FILE *fp* deklariert. Auch gilt die Folge

- Öffnen der Datei mit *fopen(),*
- Schreiben oder Lesen
- und Schließen der Datei mit *fclose()*,

allerdings mit einem wichtigen Unterschied. Ein File kann mit "w" zum Schreiben oder mit "r" zum Lesen geöffnet werden. In der Open()-Funktion des Amiga-DOS gibt es diese Unterscheidung nicht. Ein paar Fragen bleiben noch, aber die kann Listing 4.4.5 besser erklären.

```
/*
  ZCOPY.C
  Liest ein Textfile ( argv[1] ),
  setzt vor jede Zeile eine lfd. Nr.
  und schreibt nach argv[2].
  ---------------------------------- */

#include <stdio.h>
```

Listing 4.4.5.: (Fortsetzung folgende Seiten)

```
#include <string.h>
#define STRLEN 128
char string[STRLEN];
FILE *infile, *outfile;
char InName[80], OutName[80];
void Meldung(int nr);

void main( int argc, char *argv[] )
{
  int zeile = 1;
  strcpy(InName, argv[1] );
  strcpy(OutName, argv[2] );

 /* Teste auf richtige Argumente
    --------------------------  */
  if (argc != 3 )
     Meldung(1);
  if ( strcmp(InName, OutName) == 0 )
     Meldung(2);

  /* Öffne die Files und teste dabei auf Fehler
     ---------------------------------------- */
  if( (infile = fopen( argv[1], "r" )) == NULL )
     Meldung(3);
  if( (outfile = fopen( argv[2], "w" )) == NULL )
     Meldung(4);

  /* Lese und schreibe
     ---------------- */
  while( 1 )
  {
   if( fgets( string, STRLEN - 1, infile ) == NULL )
       if( feof( infile ) )
     Meldung(0);
       else
   Meldung(5);
    printf(          "%4d: %s", zeile,   string);
```

Listing 4.4.5.: (Fortsetzung folgende Seite)

```
    fprintf(outfile, "%4d: %s", zeile++, string);
  }
}
void Meldung( int nr)
{
 switch(nr)
  {
    case 0:
      printf("\n'%s' mit Zeilennumern auf '%s'
             kopiert\n", InName, OutName);
      break;
    case 1:
     puts("\nSyntax: zcopy <Qellfile> <Zielfile>");
      break;
    case 2:
      puts("\nNamen müssen verschieden sein");
      break;
    case 3:
      printf("\nKonnte Datei '%s' nicht öffnen\n",
             InName );
      break;
    case 4:
       printf("\nKonnte Datei '%s' nicht öffnen\n",
              OutName );
       break;
    case 5:
       printf("\nFehler beim Lesen von '%s'\n",
       InName );
    break;
  }
    fclose(infile);
    fclose(outfile);
    exit(nr);
}
```

Listing 4.4.5: ZCOPY versieht die Zieldatei mit Zeilennummern

Das Programm liest eine Textdatei, versieht jede Zeile mit einer laufenden Nummer und schreibt sie in eine andere Datei.

4.4.4 Die Kommandozeile, argc und *argv[]

Das Neue ist zuerst, daß die beiden Dateinamen nicht im Programm stehen, sondern beim Aufruf übergeben werden. Wenn das Programm *zcopy* heißt, und Sie die Datei *win.c* (mit Zeilennummern) auf *win.zn* kopieren wollen, tippen Sie ein

```
zcopy win.c  win.zn
```

Das Ganze nennt man die Kommandozeile, wobei *zcopy* das Kommando ist, dem die Argumente *win.c* und *win.zn* folgen. Die Auswertung ist möglich, wenn man anstatt *main()* jetzt

```
main( int argc, char *argv[] )
```

schreibt. Das sind vordefinierte Variablen. *argc* (argument count) gibt die Anzahl der Argumente an, **argv[]* (oder ***argv*) ist ein Array von Strings. In *argv[0]* steht der Programmname, in *argv[1]* das erste Argument, in *argv[2]* das zweite, usw. Weil der Programmname immer mitzählt, muß *argc* im Falle von Argumenten gleich 3 sein. Deshalb kann ich hier mit *if (argc != 3)* ganz einfach den ersten Fehler feststellen und das »Usage« ausgeben.

Zurück zu Listing 4.4.5: Die Funktion *Meldung* hat die Aufgabe, einige von den vielen Fehlermöglichkeiten auszugeben. Hier ist die Sache noch einfach. Zum Beispiel müssen Sie im Fall von *if (argc != 3) Meldung(1);* nur im *case 1* der Funktion *Meldung()* nachsehen, und wissen, daß hiermit die falsche Syntax moniert wird. In großen Programmen – oder wenn *Meldung()* in einem H-File steckt – ist es sinnvoller, Konstanten zu definieren. Gibt es z.B ein *#define SYNTAX* 1, sagt *Meldung(SYNTAX)* sofort, was gemeint ist. Für die DOS-Fehler gibt es übrigens schon Konstanten in den H-Files.

In der Hauptschleife fällt Ihnen bestimmt etwas auf. Statt *printf()* schreibe ich hier *fprintf()* (file-print), vorher hatten wir schon *gets()* und *puts()* mit dem »f« versehen. Besonders einfach ist die Sache bei *fprintf()*, wo außer dem zusätzlichen Argument für den File-Zeiger präzise die Regeln von *printf()* gelten. Tatsächlich ist der Bildschirm, bzw. das CLI-Window, im Sinne von C auch nur ein File mit einem Filepointer namens *stdout*. *printf("xxx")* ist nichts weiter als eine Kurzform von *fprintf(stdout, "xxx")*, ruft also nur diese Funktion auf.

4.4.5 Binär-File-I/O

Daraus können Sie zuerst folgern, daß *fprintf(stdout,...* schneller ist, aber das ist nicht alles. Wenn Sie Zahlen mit *fprint()* speichern, werden sie als Text aufgezeichnet. Die int-Zahl 12345 ist in der Textform eine Folge von ASCII-Zeichen, die auf der Disk auch 5 Byte belegt. Rechnerintern werden solche Zahlen binär als 2 Byte abgelegt. Große Preisfrage: Wie bekomme ich solche Zahlen im Binärformat auf die Disk und spare damit speziell bei Arrays große Mengen Platz? Die Antwort zeigt Listing 4.4.6.

```
/*BIN.C */
#include <stdio.h>

main()
{
   FILE *fp;
   int ar[10], i;

   for( i = 0; i < 10; i++ )
      ar[i] = i; /* Fülle Array */
   if( (fp = fopen( "binfile", "wb" )) != NULL )
    {
      fwrite( ar, sizeof(ar), 1, fp );
      fclose( fp );
    }
   else
      perror( "Write error" );
   if( (fp = fopen( "binfile", "rb" )) != NULL )
   {
      fread( ar, sizeof(ar), 1, fp );
      fclose( fp );
      for( i = 0; i < 10; i++ )
      printf("%d\n", ar[i] );
   }
   else
      perror( "Read error" );
}
```

Listing 4.4.6: So spart man viel Platz auf der Disk

In der fopen()-Funktion ändert sich nur der Modus in *rb* bzw. *wb*, wobei das »b« für binär steht. Neu sind die Funktionen *fread()* und *fwrite()*. Hierin bedeuten:

ar	Zeiger auf Puffer
sizeof(ar)	Größe eines Blocks
1	Anzahl der Blöcke
fp	Filepointer

Hier wird der Array *ar* mit einem einzigen *write()* auf die Disk geschrieben. Beachten Sie den Unterschied zwischen Puffer- und Blockgröße. In diesem Beispiel ist die Blockgröße gleich der Puffergröße. Vorstellbar ist aber auch ein Array von 50 Zeilen, wo man die Blockgröße auf die Zeilengröße setzt und die Blockzahl auf 50. Man hat auf diese Art viele Freiheitsgrade, zum Beispiel diesen:

```
for(i=0; i<10; i++)
  fread( &ar[10-i], sizeof(int), 1, fp );
```

Als Pufferadresse wird immer ein Array-Element angegeben, die Blockgröße reicht für eine int-Zahl. Und was passiert? Nun, auf der Disk stehen die Binärwerte der Zahlen 0, 1, 2, 3, usw. und die werden nach *ar[9]*, *ar[8]*, *ar[7]*, usw. gelesen. Endergebnis: Die Zahlen stehen in der umgekehrten Folge im Array.

4.4.6 Dynamische Puffer

Nachdem wir nun wissen, wie man sortiert und wie man Textdateien von der Diskette liest, können wir uns der praxisgerechten Lösung zuwenden. Im Gegensatz zum Listing 4.4.2 stehen nämlich die Texte nicht im Programm, sondern auf der Disk. Um sie sortieren zu können, muß man sie einlesen (logisch), aber auch den ganzen Text im Speicher halten.

```
/* TSORT_2.C
   Demo 2 zum Sortieren von Texten
*/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#define  ANZAHL 1000
```

```
#define  ZLEN 128
char *zeile[ANZAHL];

void main(int argc, char  **argv)
{
   FILE *fp;
   int i, znr, cmps();
   char buf[ZLEN];

   if (argc != 2) exit(1);

   if( (fp = fopen(argv[1], "r")) == NULL)
     exit(1);

   for(znr = 0; znr < ANZAHL; ++znr)
   {
     if ( fgets(buf, ZLEN, fp) ==  NULL )
       break;
     zeile[znr] = (char *) malloc(strlen(buf)+1);
     strcpy(zeile[znr], buf);

   }
   fclose(fp);

  /* Sortieren nach dem Quicksort-Algorithmus
  */
  qsort( (void *) zeile,  /* Ab wo sortieren    */
     (size_t)znr,         /* Anzahl Elemente    */
      sizeof(char *),     /* Anzahl Elemente    */
      cmps );             /* Vergleichsfunktion */
   for(i=0; i<znr; ++i)
     printf("%s", zeile[i] );
}
int cmps( char **elem1,char **elem2 )
{
  return strcmp(*elem1, *elem2);
}
```

Listing 4.4.7 Sortieren einer Textdatei

Doch wie groß muß der dafür zu reservierende Speicherbereich sein? Wir wollen maximal 1000 Zeilen einlesen, jede Zeile darf bis zu 128 Zeichen lang sein. Dafür einen Array *a[1000][128]* zu deklarieren, ist nicht das Ideale. Das sind nämlich 128 Kbyte, und die meisten Texte sind kleiner als 10 Kbyte. Und überhaupt gilt die Regel, daß ein Task nicht mehr Speicher anfordern sollte, als er unbedingt braucht.

Nun, wir gehen einen Kompromiß ein und belegen 4 Kbyte. Das sind 1000 Zeiger auf Strings. Die Strings gibt es noch nicht. Erst wenn eine Zeile gelesen wurde, wird mit *malloc()* ein Speicherbereich allokiert, in den genau diese Zeile paßt. Dann wird ein String-Zeiger mit der Adresse dieses Speicherbereiches geladen. Damit ist schon klar, warum die Schleife *for(znr = 0; znr < ANZAHL; ++znr)* bis 1000 läuft, genauer: so weit laufen könnte. Tatsächlich wird nach dem Lesen von null Zeichen (Datei-Ende) die Schleife mit *break* verlassen. Wenn nicht, wirkt diese Zeile:

```
zeile[znr] = (char *) malloc(strlen(buf)+1);
```

Der aktuelle String steht in *buf*, für seine Länge +1 (des Null-Bytes wegen) wird Speicher allokiert. *malloc()* gibt einen Zeiger auf diesen Bereich zurück, der Zeiger kann also auch gleich *zeile[znr]* zugewiesen werden, denn das ist ein Element des Arrays von char-Zeigern. Und was ist ein String-Array? Antwort: Ein Array von char-Zeigern. Damit hätten wir unseren Array, den wir sortieren können. Der letzte Stand von *znr* ist auch die Anzahl der Elemente.

Bliebe noch eine Frage. Muß ich denn mit *char *zeile[ANZAHL];* gleich so viele Zeiger vorhalten, wenn ich meistens weniger Zeilen lese, kann ich nicht auch die Zeiger mit *malloc()* beschaffen? Die Antwort ist ein ganz klares Jein. In diesem Beispiel muß der Zeigerarray statisch sein, nur das garantiert, daß der Array auch fortlaufend im Speicher steht. *malloc()* hingegen sucht sich den Speicher irgendwo, wo gerade Platz ist. Wollen Sie die andere Lösung, müssen Sie sogenannte Listen anwenden. Eine Liste besteht in diesem Fall aus lauter Strukturen. Jede davon hat zwei Zeiger, einen auf den Nachfolger und einen auf den String einer Zeile. Für die Struct's, hier auch Listenknoten genannt, kann man natürlich den Speicher auch mit *malloc()* anfordern.

Anhang

Die Include-Files zum Malprogramm

```
/* Include-File "stdwindow.h"
   --------------------------*/

void open_libs(void)
{
   IntuitionBase =  (struct IntuitionBase *)
               OpenLibrary("intuition.library",0L);
   if (IntuitionBase == NULL)  Exit(FALSE);

   GfxBase = (struct GfxBase *)
             OpenLibrary("graphics.library",0L);
   if (GfxBase == NULL)
    {
     CloseLibrary(IntuitionBase); /* Intu-Lib zu */
     Exit(FALSE);                 /* Ausgang     */
    }
}

struct window *open_window (short x, short y,
                            short w, short h,
                            char *name,
                            ULONG flags,
                            ULONG i_flags,
                            struct Gadget *gadget )

{
```

```
    struct NewWindow nw;

    nw.LeftEdge     = x;
    nw.TopEdge      = y;
    nw.Width        = w;
    nw.Height       = h;
    nw.DetailPen    = -1;
    nw.BlockPen     = -1;
    nw.Title        = (UBYTE *) name;
    nw.Flags        = flags;
    nw.IDCMPFlags   = i_flags;
    nw.Screen       = NULL;
    nw.Type         = WBENCHSCREEN;
    nw.FirstGadget  = gadget;
    nw.CheckMark    = NULL;
    nw.BitMap       = 0;
    nw.MinWidth     = -1;
    nw.MinHeight    = -1;
    nw.MaxWidth     = -1;
    nw.MaxHeight    = -1;

    return( (struct window *) OpenWindow(&nw) );
}


void close_all(void)
{
    CloseWindow(Window);
    CloseLibrary(GfxBase);
    CloseLibrary(IntuitionBase);
    exit(TRUE);
}
```

Das Include-File »stdwindow.h«

```
/* Include-File "gadget.h"
   ----------------------*/

APTR iadr; /* Gadget-Adresse */
int  id;   /* Gadget-ID      */
```

```
struct Gadget gadget[14];
struct IntuiText gtext[14];

/* Für String-Gadget */
struct StringInfo info;
char dobuffer[80], undobuffer[80];

/* für Prop-Gadget */
struct Image img[3];
struct PropInfo prop[3];
#define STEP (0xFFFF/0x10) /* Farbe in 16 Schritten */

/* Text in eine IntuiText-struct eintragen
   ---------------------------------------*/
void make_gtext(struct IntuiText *name, char *text,
                SHORT left, SHORT top)
{
  name->FrontPen = 1;
  name->BackPen =  0;
  name->DrawMode = JAM1;
  name->LeftEdge = left;
  name->TopEdge = top;
  name->ITextFont = 0;
  name->IText = (UBYTE *) text;
  name->NextText = NULL;
}

/* Eine Gadget-Struktur füllen
   ---------------------------*/
void make_gadget(char *gtext, struct IntuiText *gname,
                 SHORT tleft, SHORT ttop,
                 struct Gadget *gadget,
                 struct Gadget *next,
                 SHORT left, SHORT top,
                 SHORT width, SHORT height,
                 USHORT flags, USHORT activation,
                 USHORT type, APTR *render,
                 APTR  *info, USHORT id )

{
```

```
  gadget->NextGadget    = next;
  gadget->LeftEdge      = left;
  gadget->TopEdge       = top;
  gadget->Width         = width;
  gadget->Height        = height;
  gadget->Flags         = flags;
  gadget->Activation    = activation;
  gadget->GadgetType    = type;
  gadget->GadgetRender  = (APTR) render;
  gadget->SelectRender  = NULL;

  make_gtext(gname, gtext, tleft, ttop);
  gadget->GadgetText    = gname;

  gadget->MutualExclude = NULL;
  gadget->SpecialInfo   = (APTR)info;
  gadget->GadgetID      = id;
  gadget->UserData      = NULL;
}


/* Wird bei GADGETDOWN-Event aufgerufen,
   ermitttelt ID, behandelt den Fall:
   ------------------------------------ */
void do_gadget()
{

  id = (((struct Gadget *) iadr) ->GadgetID);
  switch(id)
  {
   case 1: /* Falls String-Gadget auch mit Return */
    break; /*verlassen werden soll, hier einhaken.*/

    case 2: /* "Nein im Schließen-Requester */
            break;

    case 3: do_close(); /* "Ja" im Schließen-Requ.*/
            break;

    case 4: /* Die Farbregler */
```

```
    case 5:
    case 6: SetRGB4(&Screen->ViewPort,0,
                    prop[0].HorizPot/STEP,
                    prop[1].HorizPot/STEP,
                    prop[2].HorizPot/STEP);
            break;

    case 7: /* Farben über DM-Requester */
    case 8:
    case 9:
    case 10: do_farben( (USHORT) (id - 7) );
             break;
    case 11: /* Reset-Gadget */
             SetRGB4(&Screen->ViewPort,0,0,0,10);
             prop[0].HorizPot = 0;
             prop[1].HorizPot = 0;
             prop[2].HorizPot = 10*STEP;
             RefreshGadgets(&gadget[3], Window,
                            &request[3]);
             break;
    case 12: /* OK-Gadget */
            break;

    case 13: /* "Abbruch" im Datei-Requ. */
             break;
    case 14: do_datei(); /* "OK" im Datei-Requ. */
             break;
  }
}
```

Das Include-File »gadget.h«

```
/* Include-File "requester.h"
   ---------------------------*/

/* Sie können das \ weglassen, wenn Sie das folgende
   Makro in eine Zeile schreiben
*/
#define GFLAGS ( GADGIMMEDIATE | ENDGADGET | \
                 RELVERIFY )
```

```
/* für die Rahmen der Gadgets
   -------------------------*/
SHORT cords1[] = {0,0, 282,0, 282,12, 0,12, 0,0};
struct Border border1 = {-2,-2,1,0,JAM1,5,
                         &cords1[0],NULL};

SHORT cords2[] = {0,0, 81,0, 81,21, 0,21, 0,0};
struct Border border2 = {-1,-1,1,0,JAM1,5,
                         &cords2[0],NULL};

SHORT cords3[] = {0,0, 69,0, 69,13, 0,13, 0,0};
struct Border border3 = {-1,-1,1,0,JAM1,5,
                         &cords3[0],NULL};

struct IntuiText rtext; /* Text Schließen-Requ. */

/* Eine Requester-Struktur füllen
   ------------------------------*/
void MakeReq( int i, SHORT left, SHORT top,
                     SHORT width, SHORT height,
                     struct Gadget *gadget,
                     struct IntuiText *text,
                     USHORT flags, UBYTE fill )
{
request[i].LeftEdge  = left;
request[i].TopEdge   = top;
request[i].Width     = width;
request[i].Height    = height;
request[i].ReqGadget = gadget;
request[i].ReqBorder = NULL;
request[i].ReqText   = text;
request[i].Flags     = flags;
request[i].BackFill  = fill;
request[i].ImageBMap = NULL;
}


/* Die Requester anlegen
   ==================== */
void MakeTheRequester(void)
```

```
{

/* Das String-Gadget für den File-Namen
   ------------------------------------*/
strcpy(dobuffer,"bild_0.img");
info.Buffer = (UBYTE *) dobuffer;
info.UndoBuffer = (UBYTE *) undobuffer;
info.MaxChars = 64;
info.BufferPos = 0;
info.DispPos = 0;

make_gadget("Geben Sie den Dateinamen ein:",
            &gtext[0],0,-15,
            &gadget[0], &gadget[12], 10,24,280,11,
            GADGHCOMP, STRINGCENTER | GADGIMMEDIATE,
            STRGADGET | REQGADGET,
            (APTR *) &border1, (APTR *) &info, 1);

make_gadget( "Abbruch", &gtext[12],14,7,
             &gadget[12],&gadget[13],30,50,80,20,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border2, NULL, 13);

make_gadget("OK", &gtext[13], 25,7,
             &gadget[13],NULL,190,50,80,20,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border2, NULL, 14);

MakeReq(0, 50,50, 300,85, &gadget[0], NULL, NULL, 3);


/* Der Requester für "Schließen"
   ----------------------------*/
make_gtext(&rtext, "Wollen Sie wirklich schließen?",
                    50, 7 );

make_gadget( "Nein", &gtext[1],24,7,
             &gadget[1],&gadget[2],50,40,80,20,
```

```
                GADGHCOMP, GFLAGS,
                BOOLGADGET | REQGADGET,
                (APTR *) &border2, NULL, 2);

make_gadget("OK", &gtext[2], 25,7,
                &gadget[2],NULL,190,40,80,20,
                GADGHCOMP, GFLAGS,
                BOOLGADGET | REQGADGET,
                (APTR *) &border2, NULL, 3);

MakeReq(1, 30,20, 350,70, &gadget[1], &rtext,
        NULL, 3 );


/* Der Requester für die Farbregler
   ---------------------------------*/
make_gadget("Rot", &gtext[3],-38,7,
                &gadget[3], &gadget[4],50,10,200,20,
                GADGHCOMP, GADGIMMEDIATE | RELVERIFY,
                PROPGADGET | REQGADGET,
                (APTR *) &img[0], (APTR *) &prop[0], 4);
prop[0].Flags = AUTOKNOB | FREEHORIZ;
prop[0].HorizBody = STEP;
prop[0].HorizPot = 0;

make_gadget("Grün",&gtext[4],-38,7,
                &gadget[4], &gadget[5],50,35,200,20,
                GADGHCOMP, GADGIMMEDIATE | RELVERIFY,
                PROPGADGET | REQGADGET,
                (APTR *) &img[1], (APTR *) &prop[1], 5);
prop[1].Flags = AUTOKNOB | FREEHORIZ;
prop[1].HorizBody = STEP;
prop[1].HorizPot = 0;

make_gadget("Blau",&gtext[5],-38,7,
                &gadget[5], &gadget[10], 50,60,200,20,
                GADGHCOMP, GADGIMMEDIATE | RELVERIFY,
                PROPGADGET  | REQGADGET,
                (APTR *) &img[2], (APTR *) &prop[2], 6);
prop[2].Flags = AUTOKNOB | FREEHORIZ;
```

```
prop[2].HorizBody = STEP;
prop[2].HorizPot = 10*STEP;

make_gadget( "Reset", &gtext[10],20,7,
             &gadget[10],&gadget[11],50,90,80,20,
             GADGHCOMP,  GADGIMMEDIATE | RELVERIFY,
             BOOLGADGET | REQGADGET,
             (APTR *) &border2,  NULL, 11);

make_gadget("OK", &gtext[11], 25,7,
             &gadget[11],NULL,170,90,80,20,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border2, NULL, 12);

MakeReq(2, 100,30, 300, 120, &gadget[3], NULL,
        NULL, 2);

/* Der DM-Requester für "Farben"
   ----------------------------*/

make_gadget("schwarz", &gtext[6], 3, 1,
             &gadget[6],&gadget[7],5,3,68,12,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border3, NULL, 7);

make_gadget("weiß", &gtext[7], 3, 1,
             &gadget[7],&gadget[8], 5,18,68,12,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border3, NULL, 8);

make_gadget("rot", &gtext[8], 3, 1,
             &gadget[8],&gadget[9],5,35,68,12,
             GADGHCOMP, GFLAGS,
             BOOLGADGET | REQGADGET,
             (APTR *) &border3, NULL, 9);

make_gadget("blau", &gtext[9], 3, 1,
```

```
                &gadget[9],NULL,5,53,66,12,
                GADGHCOMP, GFLAGS,
                BOOLGADGET | REQGADGET,
                (APTR *) &border3,  NULL, 10);

MakeReq( 3, 0,0, 87,75, &gadget[6], NULL,
         POINTREL | NOISYREQ,2 );

}
```

Das Include-File »requester.h«

```
/* Include-File "menu.h"
   -------------------- */

struct Menu      menu[4];
struct MenuItem  item[19];
struct IntuiText itext[19];

#define PROJEKT  0
#define FARBEN   1
#define FIGUREN  2
#define SPEZIAL  3

#define NEU      0
#define LADEN    1
#define SICHERN  2
#define ENDE     3

#define WIRKLICH 0
#define NEIN     1

#define SCHWARZ  0
#define WEISS    1
#define ROT      2
#define BLAU     3
```

```
#define   RECHTECK  0
#define  FRECHTECK  1
#define  ELLIPSE    2
#define FELLIPSE    3

#define FUELLE     0
#define MALE       1
#define REGLER     2

/* Text in eine IntuiText-struct eintragen
   ---------------------------------------- */
void make_text(struct IntuiText *name, char *text,
               SHORT left, SHORT top)
{
  name->FrontPen = 0;
  name->BackPen = 1;
  name->DrawMode = JAM1;
  name->LeftEdge = left;
  name->TopEdge = top;
  name->ITextFont = 0;
  name->IText = (UBYTE *) text;
  name->NextText = NULL;
}

/* Daten in ein Menu-Item-struct eintragen
   ---------------------------------------- */
void make_item( char *itemtext,
                struct IntuiText *name,
                struct MenuItem *item,
                struct MenuItem *next,
                USHORT left,USHORT top,
                USHORT width,USHORT height,
                ULONG flags)
{
  item->NextItem = next;
  item->LeftEdge = left;
  item->TopEdge  = top;
  item->Width    = width;
  item->Height   = height;
  item->Flags    = flags | ITEMTEXT | HIGHCOMP;
```

```
    item->MutualExclude = NULL;

    make_text(name, itemtext, 0, 0);
    item->ItemFill = (APTR)name;

    item->SelectFill = NULL;
    item->Command    = NULL;
    item->SubItem    = NULL;
}

/* Daten in einen Menu-Titel-struct eintragen
   ------------------------------------------ */
void make_menu(struct Menu *name, char *titel,
               struct Menu *next,
               struct MenuItem *first,
               USHORT left )
{
  name->NextMenu  = next;
  name->LeftEdge  = left;
  name->TopEdge   = 0;
  name->Width     = strlen(titel)*8+10;
  name->Height    = 10;
  name->Flags     = MENUENABLED;
  name->MenuName  = (BYTE *) titel;
  name->FirstItem = first;
}

/* Alle Menus anlegen
   ------------------ */
void MakeTheMenu()
{

/*---- Menu 0 --------------*/

make_item( "Neu",       /* Text des Items          */
         &itext[0],   /* Adresse IntuiText-struct */
         &item[0],    /* Adresse der Item-struct  */
         &item[1],    /* Adresse next Item        */
         5,0,         /* linke obere Ecke         */
         120,11,      /* Breite und Höhe          */
```

```
        ITEMENABLED /* Flags des Items          */
  );

make_item("Laden",    &itext[1], &item[1], &item[2],
                      5,16,120,11,ITEMENABLED);

make_item("Sichern", &itext[2], &item[2], &item[3],
                      5,32,120,11,ITEMENABLED);

make_item("Ende",     &itext[3], &item[3], NULL,
                      5,48,120,11, ITEMENABLED );

make_item("Wirklich Ende", &itext[4], &item[4],
                      &item[5], 100,8,200,11,
                      ITEMENABLED | COMMSEQ);

make_item("Lieber doch nicht", &itext[5],
                      &item[5], NULL,
                      100,24,200,11, ITEMENABLED );

item[3].SubItem = &item[4]; /* Sub-Item nachtragen */
item[4].Command =  'x';

make_menu(&menu[0],"Projekt", &menu[1],
    &item[0], 10);

/*-------- Menu 1 --------*/

make_item("Schwarz", &itext[6], &item[6], &item[7],
                      5,0,100,11,ITEMENABLED);

make_item("Weiß",     &itext[7],&item[7], &item[8],
                      5,16,100,11,ITEMENABLED);

make_item("Rot",      &itext[8], &item[8], &item[9],
                      5,32,100,11,ITEMENABLED);

make_item("Blau",      &itext[9], &item[9], NULL,
                       5,48,100,11,ITEMENABLED);
```

```
make_menu(&menu[1],"Farben", &menu[2],
                  &item[6], 120);

/*-------- Menu 2 --------*/

make_item("Rechteck",  &itext[10], &item[10],
                       &item[11],
                       5,0, 150,11, ITEMENABLED);

make_item("Gefülltes Rechteck",
                       &itext[11],
                       &item[11], &item[12],
                       5,16, 150,11, ITEMENABLED);

make_item("Ellipse",   &itext[12],
                       &item[12], &item[13],
                       5,32, 150,11, ITEMENABLED);

make_item("Gefüllte Ellipse",
                      &itext[13], &item[13],  NULL,
                       5,48,150,11,ITEMENABLED);

make_menu( &menu[2],"Figuren", &menu[3],
           &item[10], 210 );


/*-------- Menu 3 --------*/

make_item("Figur füllen",
                       &itext[14], &item[14],
                       &item[15],
                       5,0,120,11,ITEMENABLED);

make_item("Alles übermalen",
                       &itext[15], &item[15],
                       &item[18],
                       5,16, 130,11, ITEMENABLED);

make_item("Wirklich übermalen",
                       &itext[16], &item[16],
```

```
                        &item[17],
                        110,8, 150,11, ITEMENABLED
);

make_item("Lieber doch nicht",
                       &itext[17], &item[17],
                       NULL,
                       110,24,150,11, ITEMENABLED );
item[15].SubItem = &item[16]; /* Sub-Item hinzu */

make_item("Farben regeln",
                        &itext[18], &item[18],
                        NULL,
                        5,32, 130,11, ITEMENABLED );

make_menu(&menu[3],"Spezial", NULL,
                       &item[14], 300);


}

/* MENUPICK-Message auswerten und zugehörige
   Aktionen ausführen
   ---------------------------------------- */
void do_menu()
{
  switch(MENUNUM(code)) /* Switch Titel */
  {
    case PROJEKT:
       switch( ITEMNUM(code) ) /* Switch Items */
         {
           case NEU    : do_neu();
                         break;

           case LADEN : dflag = 1;
                        Request(&request[0], Window);
                        break;
           case SICHERN: dflag = 2;
                        Request(&request[0], Window);
                           break;
```

```
        case ENDE    :
          switch( SUBNUM(code) )
          {
            case WIRKLICH: do_close();
                           break;
            case NEIN    :
                           break;
          }
          break;
      }
      break;

case  FARBEN:
          do_farben( (USHORT) ITEMNUM(code) );
          break;

case  FIGUREN:
        switch( ITEMNUM(code) )
          {
            case RECHTECK : do_rechteck(0);
                            break;
            case FRECHTECK: do_rechteck(1);
                            break;
            case ELLIPSE  : do_ellipse(0);
                            break;
            case FELLIPSE : do_ellipse(1);
                            break;
          }
          break;

case SPEZIAL:
     switch( ITEMNUM(code) ) /* Switch Items */
        {
          case FUELLE: do_fuellen();
                       break;
          case MALE  :
             switch( SUBNUM(code) )
                {
                   case WIRKLICH: do_malen();
                                  break;
```

```
                        case NEIN    :
                                              break;
                }
                break;

        case REGLER: Request(&request[2], Window);
                     break;

                }
        break;

    }
}
```

Das Include-File menu.h

Glossar

Absolute Adresse: Adresse als Zahl; wird typisch einer Zeigervariablen zugewiesen.

Alert: Alarmmeldung, die typische Guru-Meldung.

Allokieren: Mit einer Funktion wie *malloc()* (memory alloc) Speicher für Variable zur Laufzeit eines Programms beschaffen.

Amiga-DOS: Disk Operating System (Betriebssystem) des Amiga.

ANSI: American Standard Institute. Amerikanischer Standard, der auch die Sprache C normt.

ANSI-Kompatibilität: Übereinstimmung mit dem ANSI-Standard.

Anwender-Gadgets: Im Gegensatz zu den System-Gadgets vom Anwendungsprogrammierer definierte Gadgets.

API: Application Program Interface. Schnittstelle zu Betriebssystemfunktionen für den Anwendungsprogrammierer.

Area: Eine Sammlung von Grafiken, die alle zusammen gezeichnet werden.

argc: »argument count«, die Anzahl der Argumente (Wörter) der Kommandozeile

Array: Eine Menge von Daten gleichen Typs.

ASCII: American Standard Code (for) Information Interchange. 256 Zahlen, die beschreiben, welche Zahl welches Zeichen auf dem Bildschirm darstellen soll. Nur die ersten 128 Zeichen sind auf allen Computern identisch.

Auflösung: Hier die Anzahl der Bildpunkte in der Horizontalen (X-Auflösung) und in der Vertikalen (Y-Auflösung).

Automatische Variable: Variable innerhalb einer Funktion, die automatisch nach dem Funktionsaufruf angelegt wird und nur so lange »lebt«, wie die Funktion läuft.

Auto-Requester: Ein Requester, der mit nur wenigen Argumenten automatisch von Intuition gezeichnet wird.

BCPL: Eine C-ähnliche Programmiersprache (genauer: ein C-Vorgänger), in der u.a. Amiga-DOS geschrieben wurde.

Befehle: Elemente der Sprache C, wie zum Beispiel *goto* oder *break*.

Bezeichner: Vom Anwender definierter Name von Variablen, Datentypen, Funktionen und Makros.

Bibliothek: Eine Sammlung von logisch zusammenhängenden Funktionen.

Binär: Zahlendarstellung nur mit 0 und 1, das Computer-interne Format.

Bit-Plane: Ein Speicherbereich, in dem jedes Bit einem Bildpunkt auf dem Schirm entspricht. Gibt es mehr als eine Bit-Plane, wird aus den Bits auf gleicher Position eine Zahl gebildet, welche die Farbe eines Bildpunktes beschreibt.

Bit-Operatoren: Operatoren wie & oder |, die Zahlen Bit für Bit nach den Regeln des logischen UND bzw. ODER verknüpfen.

BitMap: Eine Struktur, die einen Bildspeicherbereich beschreibt.

BLINK: sprich B-LINK, ein Linker, der zum Beispiel mit Lattice-C geliefert wird.

Blockgröße: Die Anzahl von Bytes, die zusammen auf die Disk geschrieben werden.

Boolean-Gadget: Ein Gadget, das entweder nur angeklickt werden kann oder nicht (im Gegensatz zu Proportional- und String-Gadgets).

Border: Eine Datenstruktur, die einen Rahmen (typisch ein Rechteck) zum Beispiel für Gadgets beschreibt.

BSS: Block Storage Segment, Datenbereich eines Programms, in dem erst zur Laufzeit Daten eingetragen werden, sog. nicht initialisierte Daten.

C-String: Ein Array vom Typ *char*, mit einem Null-Byte am Ende, welches das String-Ende markiert.

Chip-Memory: Die unteren (oder einzigen) 512 Kbyte des Amiga-RAM. Der Bereich wird auch von den Spezial-Chips des Amiga genutzt.

Code: Eine Zahl, deren Sinn erst im Kontext klar wird. Zum Beispiel bedeutet der ASCII-Code 65 den Buchstaben A.

Container: Ein Behälter für grafische Elemente. Ein Requesterist ist zum Beispiel der Container für Gadgets.

CPU: Central Processing Unit. Der Mikroprozessor, der eigentliche Rechner in einem Computer.

Dangling pointer: Ein Zeiger auf einen Speicherbereich, der nicht (mehr) dem Programm gehört.

Datenstruktur: Eine gemeinsam unter einem Namen zusammengefaßte Menge von Daten.

Debugger: Ein Werkzeug (Programm) zur Fehlersuche in Programmen.

Debug-Information: Zusätzliche und für das Programm an sich nicht erforderliche Informationen, die den Debugger unterstützen.

Deklaration: Das Definieren von Variablen.

Dekrementieren: Den Wert einer Variablen erniedrigen.

Dimensionen: Die Größe eines Arrays in einer Richtung.

Directory: Inhaltsverzeichnis von Disketten oder Festplatten.

Direktive: Eine Anweisung an den Compiler, zum Beispiel mit *#include*, eine Datei einzubeziehen.

Dopppelte Indirektion: Ein Zeiger zeigt auf einen weiteren Zeiger, der dann erst auf ein Objekt zeigt.

Draw: Englisch zeichnen, Teil vieler Grafikfunktionen.

Dynamische Puffer: Speicherbereiche, die erst zur Laufzeit des Programms angelegt werden.

Editieren: Die Eingabe und das Ändern von Text.

EOF: End of File: Das Ende einer Datei.

Ereignisse: Bedienaktionen, wie die Betätigung der Tastatur, der Maus oder ein Diskette neinschub/-auswurf.

Escape-Zeichen: In C das %-Zeichen mit der Bedeutung, daß die folgenden Zeichen nicht zum Text gehören, sondern Formatier- und Steuerzeichen sind.

Events: Siehe Ereignisse.

Exec: Der (u.a.) für das Multitasking zuständige Teil des Amiga-Betriebssystems.

Externe Variable: Eine außerhalb aller Funktionen deklarierte Variable, die für alle Funktionen und andere Module zugänglich ist.

FALSE: Logisch falsch, in C als Null definiert.

Fast Memory: Speicherbereich über 512 Kbyte, der deshalb schnell (fast) ist, weil sich die CPU ihn nicht mit den Spezial-Chips teilen muß.

FIB: Kürzel für File-Info-Block, eine Datenstruktur mit Kenngrößen eines Files (Datei).

File: Englisch für Datei.

Filepointer: Ein Zeiger, der auf das nächste zu lesende Zeichen in einer Datei zeigt bzw. auf die Position, ab der geschrieben werden soll.

Flag-Bits: Bits innerhalb eines Datenwortes, denen eine bestimmte Bedeutung zugewiesen ist.

Flood: Fluten, Füllen eines Grafikbereiches mit einer Farbe.

Font: Englisch für Zeichensatz.

forbid: Verbiete. Hier Zugriff anderer Tasks, praktisch Ausschalten des Multitasking.

FOREVER: Ausdruck für eine Endlosschleife, meistens als *for(;;)* realisiert, oft auch als Makro definiert.

Format-Code: Die Formatierungs-Codes in *printf()* und *scanf()*.

Gadget: Im amerikanischen Slang »Dingsbums«, hier ein grafisches Bedienelement (z.B. Schalter).

Grafik-Cursor: Die unsichtbare Position in der Größe eines Bildpunkts, von der die nächste Zeichenoperation ausgeht.

Icon-Editor: Programm zum Erstellen und Ändern von Icons (Sinnbildern).

IDCMP: Intuition Direct Communication Message Port. Datenstruktur, über welche die Kommunikation zwischen Intuition und einem Window läuft, bzw. dem Programm, dem das Window gehört.

Include-Anweisung: Anweisung an den Compiler, eine Datei in die Übersetzung einzubeziehen.

Index-Variable: Eine Variable, die genutzt wird, um verschiedene Array-Elemente anzusprechen.

Initialisieren: Einer Variablen einen Anfangswert zuweisen.

Inkrementieren: Den Wert einer Variablen erhöhen.

Intuition: Die grafische Bedienoberfläche und das API (siehe dort) des Amiga.

Item: Hier ein Menüpunkt

Iteration: Durchlauf (durch eine Schleife).

JAM1: Ein Zeichenmodus des Amiga, hier überschreibend.

JAMw: Ein Zeichenmodus des Amiga, hier transparent.

K&R: Kernighan und Ritchie: Erfinder der Sprache C und gleichzeitig Name für den alten C-Standard (neu ist ANSI).

Kommando-Taste: Tastenkürzel für einen Menüpunkt.

Kommandozeile: Der Programm-Name und die folgenden Argumente, wie man sie im CLI eintippt.

Kommentar: Erklärender Text in einem Programmtext, der vom Compiler ignoriert wird.

Komponenten-Operator: Das Punkt-Zeichen oder »->« zum Zugriff auf die Komponenten einer Datenstruktur.

Konstante: Name für eine Zahl, ein Zeichen oder Text, der im Programmlauf nicht geändert wird.

Laufvariable: Variable, die in einer Schleife hoch- oder abwärts gezählt wird und meistens beim Erreichen eines Endwertes zum Abbruch der Schleife führt.

Library: Siehe Bibliothek.

Linken: Verschiedene bereits kompilierte Module und Bibliotheksfunktionen zusammenbinden.

Linker: Programm, welches das Linken (s.o.) ausführt.

Lock: Sperre, die verhindert, daß mehrere Tasks gleichzeitig auf eine Datei zugreifen.

Logik-Fehler: Vom Compiler nicht erkennbarer Fehler, der zur Fehlfunktion oder zum Abbruch eines Programms führt.

Makro: Ein kurzer Name für einen längeren Text, der immer dann vom Compiler eingesetzt wird, wenn der Makroname im Programmtext auftaucht.

Mehrfachproblem: Die Tatsache, daß in einem Array oder einer anderen Datenstruktrur mehrere gleiche Einträge vorhanden sind, die es zu erkennen und ggf. zu eliminieren gilt.

Mehrfachverzweigung: Das Programm soll in Abhängigkeit vom Wert einer Variablen an verschiedenen Stellen fortgesetzt werden (in C mittels *switch()*).

MenuStrip: Menüstreifen oder Menüleiste, die beim Druck auf die rechte Maustaste erscheint.

Message: Nachricht oder Botschaft, die eine Task an eine andere sendet. Praktisch werden dafür Daten in eine reservierte Struktur namens Message-Port geschrieben.

Nibble: Ein halbes Byte (4 Bit).

Oktal: Ein Zahlensystem auf der Basis 8.

Pixel: Englisches Kürzel für »Picture Element«, ein Bildpunkt.

Proportional-Gadget: Ein Gadget in Form eines Schiebereglers.

Prototyp: Eine Funktionsdeklaration mit den Namen und Typen der Parameter.

Qualifier: Hier ein Code, der Tasten wie [Shift] und [Ctrl] beschreibt.

Requester: Ein Container (meistens ein Rechteck) für mehrere Gadgets, die dann alle auf einmal erscheinen bzw. verschwinden.

ROM-Font: Ein im ROM abgelegter Zeichensatz.

ROM-Grafik: Im ROM abgelegte Grafik-Funktionen, typisch besonders schnelle Grundroutinen.

Schlüsselwörter: Reservierte Wörter einer Programmiersprache, die nicht als Bezeichner verwendet werden dürfen.

String: Zeichenkette. Eine Folge beliebiger Zeichen, die in C mit einem Null-Zeichen abgeschlossen sein muß.

String-Literal: Ein beliebiger Text in Anführungszeichen, wird von C als String-Konstante betrachtet.

Sub-Item: Ein Untermenüpunkt.

Turn-Around-Zeit: Zeit vom Verlassen des Editors über Dauer von Compiler- und Linkerlauf bis zum Wiedereintritt in den Editor.

Variable: Ein symbolischer Name für einen Speicherplatz, in dem immer neue Daten abgelegt werden können.

Verzweigung: Fortsetzung des Programms an einer anderen Stelle als beim nächstfolgenden Befehl.

Zeiger: Die Adresse einer Variablen bzw. eines Speicherplatzes.

Stichwortverzeichnis

A

Abs 229
Absolute Adresse 201
ACCESS_READ 180
ACCESS_WRITE 180
ACTIVATE 105
Activation-Flags 130
AddGadget 143, 230
Adressen, 56
Alarm 154
Alert 154
AllocMem 177, 178
AllocRemember 214, 231
Allokieren 177
Amiga-DOS 176, 178, 224
Amiga.lib 222, 224
ANSI 43
ANSI-Kompatibilität 199
Anwender-Gadgets 128
AreaEllipse 164
AreaEnd 165
Argc 258
Argv 258
Arithmetische Operatoren 84
Array 54, 58
Array deklarieren 57
Array-Index 55, 192
Assembler-Quelltext 196
Atof 229
Atoi 229
Auflösung 188
Auto-Requester 186
AUTOKNOB 130
AutoRequest 149, 231
Aztec C 20, 31, 227
Aztec-Compiler 196
Aztec-Linker 197

B

Backfill 148, 153
BCPL 178
Befehle 38
Bibliotheken 201
Binär-File-I/O 259
Binär-Files 254
Bit-Operatoren 87
Bit-Planes 186
BitMap 186
BLINK 198
Blockgröße 260
Boolean-Gadget 128
Border 153
Break 90, 126
BSS 198 f.
BuildSysRequest 231

C

C-Array 192
C-String 55
Chip-Memory 197, 199, 209, 213
Class 100
ClearDMRequest 146, 232
ClearMenuStrip 126, 232
ClearPointer 209, 232
Close 238

CLOSEWINDOW 99, 106, 127, 232
Code 100
Command-Key 113
COMMSEQ 117
Compile-/Link-Beispiele 51
Compiler 13
Compiler/Linker-Aufruf 224
COMPLEMENT 216
Container 128, 145
Continue 95
Cos 229

D

Dangling pointer 194 f.
Datenstruktur 60, 70
Datentyp 39
DEADEND_ALERT 154
Debug-Information 197, 199
Define 38
Deklaration 58
Deklarieren 47, 61
DeleteFile 238
Deutsche Tastatur 22
Dimensionen 58
Directory durchsuchen 183
Direktive 37
DISKREMOVED 150
DisplayAlert 232
DisplayBeep 232
Div 229
Do 90, 92
Do...while 92
Doppelte Indirektion 201
Double-Menu-Requester 146 f. 154
Draw 163, 236
DrawCircle 237
DrawEllipse 164, 237
Dynamische Puffer 260

E

Editieren 50
Ellipse 164
Else 88
ENDGADGET 147, 153
EndRequest 232
EOF 176
Ereignisse 99
Escape-Zeichen 46
Events 99, 162
Examine 183
Exclusive Write Lock 174
Exec 100, 200
ExecMessage 100
Execute 185
Exit 79, 229
ExNext 183

F

FALSE 92
Fast-Memory 198
Fclose 255
Festplatte 226
FGO 206
FIB 181
FILE 255
File-Handle 175
File-I/O in Standard-C 254
File-Nummer 175
File-Zeiger 255
FileHandle 175
FileInfoBlock 181
FileLock 180
Filepointer 255
FirstGadget 143
Flag-Bits 83
Flags 130
Flood 237
Font 219
Fopen 255, 260
For 90, 93 f.
For-Schleife 94, 197
Forbid 203
FOREVER 94
Format-Code 46
Formatstring 76
Fprintf 259
Fread 260
FREEHORIZ 130

FreeRaster 165
FreeRemember 202, 215
FreeSysRequest 233
Funktionen 37
Funktions-Ergebnisse 202
Fwrite 260

G

Gadget-Liste 143
Gadget-Typen 128
Gadgets 70, 83, 128
GADGIMMEDIATE 130
GetDefPrefs 233
GetMsg 162, 205
GetPrefs 233
GetRGB4 188
Gets 229
Gfx.h 104
Gleichheitszeichen 191
Goto 91
Grafik-Cursor 105

H

Hardware-Voraussetzungen 16
HIBOX 117
HICOMP 117
Hochsprache 13
Hochstriche 49
HorizBody 138
HorizPot 138

I

I/O 98
IAddress 101
Icon-Editor 208
IconED 208
IDCMP 99, 205
IDCMP-Flag 99, 106, 109
IDCMP-Flags für Requester 147
IDCMPWindow 101
If 86, 92
If-Bedingung 88
Image 128
Include 72
Include-Files 38, 72
Index 54, 193
Indexvariablen 55
Info-Struktur 138
Initialisieren 47, 58, 61
Inkrement- und Dekrement-Operatoren 85
Input 99, 238
Installation 20
IntuiMessage 100, 125
IntuiTextLength(itext) 233
INTUITICK-Event 110
INTUITICKS 106
Intuition 99
Intuition.library 78
IntuitionBase 73, 78
INVERSVID 216
IoErr 184, 238
Isalnum 230
Isalpha 230
Isdigit 230
Islower 230
Isprint 230
Ispunct 230
Isspace 230
Isupper 230
Item-Liste 114
ITEMENABLED 117
ITEMNUM 126
Items 113 f.
ITEMTEXT 117
Iteration 96
ITextFont 216

J

JAM1 216
JAM2 216

K

K&R 43
Klammern 192
Komma-Operator 95
Kommando-Taste 113
Kommandozeile 258

Kommentare 37
Kompakte Programme 222
Kompilieren 50
Komponenten-Operator 68
Konstante 49

L

Lattice C 25, 32, 227
Lattice-Compiler 198
Laufvariable 94
Libraries 72, 201
Library 72
Linken 50
– mit float 51
Linker 13, 50
Lock 175, 180
Lock-Struktur 180
Locks 174
Log 230
Logik-Fehler 88
Logische Operatoren 87
Lottozahlen 253
LSE 31

M

Main() 222
Main-Funktion 40
Makro 247
Makros 125 f., 160, 225, 250
Malloc 195, 202
Maus zeichnen 163
Maustaste 163
Mauszeiger 209
Mehrfachproblem 253
Mehrfachverzweigungen 88
MEMACS 30
MEMF_CHIP 177, 215
MEMF_CLEAR 177
Menü 113, 115
MENUENABLED 116
MenuItem 116 f.
MenuName 116
MENUNUM 126
MenuOff 116
MenuOn 116
MENUPICK 127, 162
MenuStrip 113 f.
Message-Ports 99
Micros 101
MODE_NEWFILE 175
MODE_OLDFILE 175
ModifyIDCMP 233
MOUSEBUTTONS 164
MOUSEMOVE 99, 106, 164
MouseX 101
MouseY 101
Move 105, 163, 237
MoveWindow 234
Multitasking 174, 201

N

NEWSIZE 106
NewWindow 71, 74, 82, 105
NextItem 116
Nibbles 188
NOISYREQ 145, 149, 154
Null-Byte 55, 193

O

OffGadget 234
OffMenu 234
OFFSET_BEGINNING 239
OFFSET_CURRENT 239
OFFSET_END 239
Oktal 50
OnGadget 234
OnMenu 234
Open 238
OpenDiskfont 220
OpenFont 220
OpenLibrary 72
OpenWindow 71 f., 78, 82, 234
Operatoren 12
Output 99, 239

P

Permit 203
POINTREL 147, 148, 154

Pow 230
PREDRAWN 148
Printf() 45
PrintIText 234
Programm-Icon 208
PROPGADGET 138
Proportional-Gadget 128, 131
Protos 223
Prototypen 44, 161
Punkt-Operator 68
Putc 230
Puts 230

Q

Qsort 247
Qualifier 101
Quick-Sort 247

R

Rand 250
RASSIZE 186
RastPort 104, 106
RAWKEY 106
Read 178, 239
RECOVERY_ALERT 154
RectFill 164, 237
RefreshGadgets 143, 235
Relative Requester 147
RelLeft 147
RelTop 147
RELVERIFY 130
RememberKey 214
Remove 230
RemoveGadget 143, 235
Rename 239
ReplyMsg 99, 109, 205
REPORTMOUSE 105, 235
REQACTIVE 149
REQCLEAR 147
REQGADGET 147
REQOFFWINDOW 149
REQSET 147
Request 145 f., 235
Requester 145, 147, 161
ROM-Font 220
ROM-Grafik 164

S

Scanf 86, 195
Scanf() 59
Schachtelungen 37
Schleifen 90
Schlüsselwörter 38
Schreiben von Files 175
Seconds 101
Seek 239
SetAPen 105, 237
SetBPen 237
SetDMRequest 146, 154, 235
SetDrPt 237
SetMenuStrip 127, 235
SetPointer 209, 212, 236
SetPrefs 236
SetRGB4 138
Shared Read Lock 174
Short int 58
Sichtbarkeit von Variablen 40
Signal-Bit 107
Sizeof 250
SizeWindow 143, 236
Software-Voraussetzungen 17
Sortieren 247, 250
SpecialInfo 130
SpecialLink 100
Speicher dynamisch 177
Sprintf 110
Sqrt 230
Standard-C-Funktionen 229
Standard-I/O-Kanäle 222
Status von Disketten 178
Stdwindow.h 80
Stilarten 215
Strcat 60, 230
Strcmp 230
Strcpy 55, 78
Strcspn 230
String-Gadget 128, 130, 137
String-Literal 74

STRINGCENTER 130
StringInfo 130
Stringliteral 33, 38
Strings 59, 65, 193
Stringvergleichsfunktion 93
Strncat 230
Strstr 184
Struct 60
– BoolInfo 243
– Border 244
– Gadget 242
– Image 245
– IntuiMessage 245
– IntuiText 244
– Menu 242
– MenuItem 242
– NewScreen 240
– NewWindow 241
– Preferences 245
– PropInfo 243
– Remember 245
– Requester 243
– Screen 240
– StringInfo 244
– Window 241
Struktur 36, 63
Struktur-Komponenten 82
Strukturierte Variablen 53, 61
Strukturtyp 60
Sub-Item 113
SUBNUM 126
Suchstring 184
Switch 88, 90
SYSREQUEST 149
System Request 150
System-Requester 146
System-Ressourcen 203
System-Routinen 200

T

Tan 230
Text 104 f., 110, 238
Text-Attribute 219
TextAttr 219
Texteingabe 220
Textfiles 254
Time 111
Time.h 104
Tolower 230
TOPAZ_EIGTHY 220
TOPAZ_SIXTY 220
Toupper 230
TRUE 92
Turn-Around-Zeit 50
Typ 39
Type-Casting 74, 76

U

Ungleichoperator 192
UserPort 99
UserPort-Struktur 107

V

VANILLAKEY 106, 164, 221
Variable 39
Verzweigungen 84
Vorkompilierte Header-Files 226

W

Wait 80, 106, 162, 205
Warnings 197
WBENCHSCREEN 71
While 90, 92
While-Bedingung 93
Window-Flag 105
Window-Struktur 71
WINDOWCLOSE 105
WINDOWDEPTH 105
WINDOWDRAG 105
WindowLimits 236
WindowPort 99
Windows 70
WINDOWSIZING 105
WindowToBack 236
WindowToFront 236
Workbench-Programme 208
Write 178, 186, 239
WritePixel 163, 238

Z

Z-Editor 30
Zeichen-Array 54
Zeichensätze 215
Zeichenvariablen 53
Zeiger 56, 64 f., 71, 74, 86, 194
Zeiger auf Strukturen 66
Zeigerarithmetik 64
Zeigerarray 262
Zeigertypen 77
Zufallszahl 247, 250
Zweidimensionales Array 58

Markt&Technik **W O R K S H O P** **AMIGA**

Erfolgreich starten – sicher nutzen.

C auf dem Amiga

Zum Programm:

Wer den Amiga möglichst systemnah programmieren möchte, verwendet in der Regel die Programmiersprache C. Lattice C und Aztec C sind die üblichen Compiler; beide erfüllen alle Voraussetzungen nach dem ANSI-C-Standard.

Zum Buch:

Workshop – das heißt: sichere Lernerfolge in kürzester Zeit.

Erfolgreich starten

Im Einführungskapitel erfahren Sie alles, was zum grundlegenden Umgang mit C und mit dem Buch notwendig ist. Außerdem: Starthilfe in Sachen Installation. Das Tutorium fährt Sie dann in zehn Sitzungen in diese interessante und leistungsfähige Programmiersprache ein – ausführlich und praxisgerecht – anhand eines unterhaltsamen und nützlichen Projekts. Das Ergebnis: ein eigenes Malprogramm mit Windows, Menüs und Requestern.

Sicher nutzen

Das Kapitel Know-how wird Ihr Ratgeber für die tägliche Praxis: häufige Fehlerquellen und entsprechende Lösungsvorschläge; eine umfangreiche Sammlung nützlicher Tips & Kniffe und vieles mehr.

Dann die Referenz: knapp und präzise – alle Befehle auf einen Blick. Als nützliches Add-on: eine handliche Befehlskarte mit dem Allerwichtigsten. Der Abschnitt Schnellinformation nach dem Motto »Wie programmiere ich ...?« bietet schließlich eine Reihe ausgewählter Tools zu häufigen Aufgabenstellungen der Praxis.

Workshop – verständlich und informativ; zu populären Themen wie Grafik, Musik, Textverarbeitung und Programmiersprachen.

Herausgegeben von .TXT Redaktionsteam Baumann & Partner. Ein Team aus Fachlektoren und Spezialisten, die aus eigener EDV- und Verlagspraxis die Informationsbedürfnisse von Computeranwendern kennen.

Systemvoraussetzungen:

Hardware:
Commodore Amiga mit zwei Diskettenlaufwerken und mindestens 512 Kbyte Arbeitsspeicher, empfehlenswert 1 Mbyte RAM

Software:
Aztec C oder Lattice C der Version 5.x

ISBN 3-87791-026-2

9 783877 910269 03900

DM 39,– sFr 37,– öS 304,–

Markt&Technik